Pisin yhteinen jatke -ongelman ratkaiseminen koodaavan tietorakenteen avulla

Mikko Määttä

Helsinki 30.8.2018

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen osasto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen osasto

Tekijä — Författare — Author
Mikko Määttä

Työn nimi — Arbetets titel — Title

Pisin yhteinen jatke -ongelman ratkaiseminen koodaavan tietorakenteen avulla

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	30.8.2018	54 sivua

Tiivistelmä — Referat — Abstract

Pisin yhteinen jatke -ongelmassa tarkoituksena on selvittää tietorakenteen avulla merkkijonon kahden loppuosan pisimmän yhteisen alkuosan pituus. Ongelman nopea ratkaiseminen on tärkeää esimerkiksi monissa merkkijonoalgoritmeissa. Lisäksi tiedon määrän jatkuva kasvu lisää tarvetta minimoida tietorakenteen viemä tila.

Tutkielmassa käsitellään pisin yhteinen jatke -ongelman ratkaisemista koodaavalla tietorakenteella. Tällöin alkuperäiseen merkkijonoon päästään käsiksi vain ennalta määriteltyjen kyselyjen avulla, eikä merkkijonoa tarvita kyselyissä. Tyypillisesti koodaava tietorakenne vie vähemmän tilaa ja sisältää vähemmän informaatiota kuin alkuperäinen tieto.

Taustan antamiseksi käsitellään ensin pisin yhteinen jatke -ongelman perinteisiä ratkaisuja. Sen jälkeen tarkastellaan tiedon tilavaativuutta informaatioteoreettisen entropian avulla, mikä luo perustan arvioida tietorakenteiden tilavaativuuden optimaalisuutta ja aika–tila-vaihtokauppaa. Lisäksi annetaan esimerkkejä koodaavan tietorakenteen käytöstä ja ongelman tilaa säästävistä ratkaisuista.

Yksityiskohtaisesti käsitellään pisin yhteinen jatke -ongelman ratkaisua, jossa käytetään koodaavaa tietorakennetta. Ratkaisussa on kaksi pääasiallista osaa, joiden avulla vastaus selvitetään. Toteutuksessa käytetään hyväksi useita tietorakenteita, esimerkiksi loppuosapuuta, de Bruijn -verkon muunnosta ja virittävää puuta. Algoritmin ja tietorakenteen toteutuksen lisäksi tarkastellaan tietorakenteen tilavaativuuden ylärajan parantamista ja puiden esittämistä toteutuksessa tilaa säästäen.

Keskeiset johtopäätökset ovat seuraavat. Useita kyselyjä tehtäessä pisin yhteinen jatke -ongelma voi kannattaa ratkaista tietorakenteen avulla. Tiedon määrän kasvun ja tietorakenteiden kehityksen seurauksena ongelmaan on viime vuosina esitetty tilaa säästäviä ratkaisuja. Niissä optimoidaan aika- ja tilavaativuutta monin eri tavoin. Ongelmaan on olemassa muun muassa vakioaikainen, koodaavaa tietorakennetta hyödyntävä ratkaisu, jolla voidaan päästä alilineaariseen tilavaativuuteen ilman alkuperäisen merkkijonon korkeaa tiivistyvyyttä. Viimeaikaisten ratkaisujen suorituskyvystä sovelluksissa ei ole vertailutietoa.

ACM Computing Classification System (CCS):
Theory of computation → Data compression
Theory of computation → Pattern matching

Avainsanat — Nyckelord — Keywords
pisin yhteinen jatke, koodaava tietorakenne, entropia, merkkijonohaku

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpulan tiedekirjasto

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Tallennetun tiedon määrä tunnetusti kasvaa nopeasti. Tietoa kerätään valtavasti luonnonilmiöistä ja yhteiskunnan aloilta, esimerkiksi tieteen tutkimuskohteista, ihmisten viestinnästä ja liike-elämästä. Runsaasti tallennetaan myös kirjoja, uutisia, elokuvia ja musiikkia.

Tiedon määrä voi muodostua ongelmaksi jo aiheiltaan suppeissa tehtävissä, esimerkiksi säähavaintojen tai ihmisen perimän analysoinnissa. Ripeää käsittelyä varten tieto on usein pidettävä keskusmuistissa, josta tiedon hakeminen on noin 10^5 kertaa nopeampaa kuin massamuistista [40]. Keskusmuisti ja nopeat välimuistit ovat suhteellisen pieniä verrattuna massamuisteihin ja nykyiseen tiedon määrään. Lisäksi tehokkaat haut suuresta tietomäärästä vaativat tyypillisesti indeksointia jonkin tietorakenteen avulla, mikä lisää tilantarvetta.

Tiedon suuresta määrästä aiheutuvia ongelmia voi helpottaa sopivilla algoritmeilla, jotka käyttävät massamuistia tehokkaasti, minimoivat saman tietoalkion lukukertojen määrän tai rinnakkaistavat tiedon käsittelyn. Mikään lähestymistapa ei ratkaise kaikkia ongelmia, koska tehtävät, käytössä oleva muistin määrä ja laskentateho vaihtelevat. Tiedonhallintaa auttaa myös tiedon tiivistys. Yleensä tiivistetty tieto on purettava ennen käsittelyä, eikä tiivistys silloin ratkaise tiedonkäsittelyn aikaisia ongelmia. Tilaa säästävät tietorakenteet sen sijaan tukevat haluttuja operaatioita ilman purkamista ja vievät myös vähemmän tilaa kuin vastaavat perinteiset rakenteet. Operaatiot ovat monesti käytännössä hitaampia kuin perinteisillä rakenteilla, mutta kokonaisuutena tehtävän suoritus voi nopeutua, jos tilaa säästävä tietorakenne mahtuu keskusmuistiin massamuistin sijasta.

Operaatioihin ei aina tarvita kaikkea alkuperäistä informaatiota. Esimerkiksi lämpötilataulukosta saatetaan kysyä osavälin minimi- tai maksimiarvon indeksiä lämpötilan sijasta. Pisin yhteinen jatke -ongelmassa puolestaan selvitetään merkkijonon kahden loppuosan pisimmän yhteisen alkuosan pituus, eikä vastaukseen tarvita itse alkuosaa. Tällöin tilantarvetta voi mahdollisesti minimoida käyttämällä koodaavaa tietorakennetta. Se sisältää yleensä vähemmän informaatiota kuin alkuperäinen tieto mutta tukee ennalta määriteltyjä kyselyjä ilman pääsyä alkuperäiseen tietoon.

Jäljempänä merkintä $A[1..n]$ tarkoittaa n alkion taulukkoa tai merkkijonoa ja merkintä log kaksikantaista logaritmia. Indeksointi alkaa ykkösestä. Aika- ja tilavaativuuksien tavanomaiset O-, Ω- ja Θ-notaatiot tarkoittavat vastaavasti algoritmin asymptoottista ylärajaa, alarajaa ja ala- ja ylärajaa syötteen koon funktiona. Laskennan malli on hajasaantilaitteisto (*random access machine*). Siinä tavanomaiset aritmeettiset operaatiot ja bittejä käsittelevät loogiset operaatiot ovat vakioaikaisia konesanalle. Konesanan kokoinen alkio voidaan lukea muistista ja kirjoittaa muistiin vakioajassa. Oletetaan, että konesanaan mahtuvalla osoittimella voi viitata mihin tahansa käsiteltävään alkioon. Tilavaativuuden $O(n)$ konesanaa oletetaan vastaavan $O(n \log n)$ bittiä.

Tässä tutkielmassa käsitellään pisin yhteinen jatke -ongelman ratkaisemista koodaavan tietorakenteen avulla. Pisimmän yhteisen jatkeen etsintä on keskeinen osaongel-

ma monissa merkkijonohauissa. Kiinnostus koodaaviin tietorakenteisiin puolestaan on lisääntynyt viime vuosina tietomäärien kasvaessa. Ensimmäinen koodaavaa tietorakennetta käyttävä ratkaisu ongelmaan julkaistiin vuonna 2017.

Tiivistelmä johtopäätöksistä on, että useita kyselyjä tehtäessä pisin yhteinen jatke -ongelma kannattaa monesti ratkaista tietorakenteen avulla. Tiedon määrän kasvun ja kompaktien tietorakenteiden kehityksen seurauksena ongelmaan on viime vuosina esitetty tilaa säästäviä ratkaisuja. Niissä optimoidaan aika- ja tilavaativuutta monin eri tavoin. Ongelmaan on olemassa muun muassa vakioaikainen, koodaavaa tietorakennetta käyttävä ratkaisu, jolla voidaan päästä alilineaariseen tilavaativuuteen ilman alkuperäisen merkkijonon korkeaa tiivistyvyyttä. Viimeaikaisten ratkaisujen suorituskyvystä sovelluksissa ei ole vertailutietoa.

Luvussa 2 esitellään pisin yhteinen jatke -ongelma sovelluksineen ja ongelman perinteisiä tietorakenteita käyttävät ratkaisut. Luvussa 3 käsitellään informaatioteoreettisen entropian avulla tiedon tilavaativuutta, mikä luo perustan arvioida tietorakenteiden tilavaativuuden optimaalisuutta ja aika–tila-vaihtokauppaa. Luvussa 4 tarkastellaan tilaa säästävien tietorakenteiden toteutusta käyttäen esimerkkeinä osavälikyselyä ja pisin yhteinen jatke -ongelman ratkaisuja. Luvussa 5 esitellään koodaavaa tietorakennetta käyttävä ratkaisu ongelmaan. Luku 6 sisältää johtopäätökset.

2 Pisin yhteinen jatke

Luvussa määritellään pisin yhteinen jatke -ongelma ja kerrotaan sen sovelluksista. Sen jälkeen käydään läpi tilavaativuudeltaan perinteistä loppuosataulukkoa ja -puuta käyttävät ratkaisut ongelmaan. Näin saadaan vertailukohta tilaa säästäviä tietorakenteita varten, ja samalla esitellään nämä myöhemminkin vastaan tulevat rakenteet.

2.1 Ongelma ja sovellukset

Pisin yhteinen jatke -ongelmassa eli LCE-ongelmassa (*longest common extension*) tavoitteena on selvittää merkkijonon kahden loppuosan pisimmän yhteisen alkuosan pituus [50]. Olkoot $S[1..n]$ merkkijono ja $1 \leq i, j \leq n$. Määritellään, että kysely $\texttt{lce}_S(i, j)$ palauttaa loppuosien $S[i..n]$ ja $S[j..n]$ pisimmän yhteisen alkuosan pituuden. Triviaali erikoistapaus on $i = j$, jolloin $\texttt{lce}_S(i, j) = n - i + 1$.

Ongelman muunnelmassa loppuosat ovat eri merkkijonoissa $S[1..n]$ ja $T[1..m]$, jolloin $1 \leq i \leq n$ ja $1 \leq j \leq m$. Muunnelma palautuu alkuperäiseksi ongelmaksi tarkastelemalla katenoitua merkkijonoa $S\$T$, missä merkki \$ ei esiinny kummassakaan merkkijonossa. Vastauksen antaa kysely $\texttt{lce}_{S\$T}(i, |S\$| + j)$.

LCE-ongelma esiintyy osaongelmana useissa merkkijonohaun muunnelmissa. Likimääräisessä haussa etsitään merkkijonon S kohdat, joihin päättyvän osajonon muokkausetäisyys hahmoon $P[1..m]$ on enintään k merkkiä. Kahden merkkijonon muok-

kausetäisyydellä tarkoitetaan yksittäisten merkkien lisäysten, poistojen tai korvaamisten vähimmäismäärää, jolla toisen merkkijonon voi muuntaa toiseksi. Arvolla $k = 0$ kyse on tarkasta hausta. Edelleen voidaan hakea kahden merkkijonon pisintä yhteistä osajonoa. LCE-ongelma liittyy myös erilaisten säännönmukaisuuksien etsintään: Merkkijonosta saatetaan hakea tietyin välein toistuvaa osajonoa, palindromeja tai tietyin välein toistuvia palindromeja. Palindromeja voi etsiä kaksiulotteisestakin tekstistä.

Katsotaan esimerkkinä LCE-ongelman esiintymistä yksinkertaisessa tarkassa haussa. Olkoot $S[1..7] = \texttt{abababb}$ ja siitä etsittävä hahmo $P[1..5] = \texttt{ababb}$. Aloitetaan haku vertaamalla merkkijonojen S ja P yksittäisiä merkkejä toisiinsa alkaen ensimmäisistä merkeistä. Viidensille merkeille pätee $P[5] \neq S[5]$, joten hahmoa ei löytynyt. Naiivi algoritmi siirtäisi nyt hahmoa yhden merkin verran oikealle ja aloittaisi uuden iteraation vertaamalla aluksi merkkejä $P[1]$ ja $S[2]$. Mutta jo hahmon rakenteesta selviää, ettei hahmoa löydy kohdasta $S[2]$ alkaen, koska neljä ensimmäistä merkkiparia ovat täsmänneet mutta $P[1] \neq P[2]$. Toisaalta hahmo kannattaa siirtää kaksi merkkiä oikealle alkamaan kohdasta $S[3]$, koska $P[1..2] = P[3..4]$. Lisäksi nyt tiedetään hahmon ja osajonon $S[3..7]$ pisimmän yhteisen alkuosan pituuden olevan vähintään kaksi. Siis hahmon etsintää, jonka ohessa selviää pisimmän yhteisen alkuosan pituus, voi jatkaa vertaamalla seuraavaksi merkkejä $P[3]$ ja $S[5]$.

Tällaisia havaintoja hahmon rakenteesta ja pisimmistä yhteisistä alkuosista käytetään hyväksi esimerkiksi Knuth–Morris–Pratt-algoritmissa [35] ja muissa hahmoa esikäsittelevissä hakualgoritmeissa. Tarkan merkkijonohaun voi myös kokonaan palauttaa LCE-ongelmaksi tarkastelemalla katenoitua merkkijonoa $S\$P$. Hahmo P löytyy alkaen kohdasta $S[i]$, jos ja vain jos $\texttt{lce}_{S\$P}(i, |S\$| + 1) = |P|$, missä $1 \leq i \leq |S| - |P| + 1$.

LCE-ongelma ilmenee Lempel–Ziv-tekijöihinjaossa. Olkoon $S[1..n] = f_1 f_2 \dots f_z$, missä $f_1 = S[1]$. Osajonot $f_1, \dots, f_z$ ovat merkkijonon S Lempel–Ziv 77 -tekijät [56], jos kaikilla $1 < i \leq z$ pätee, että

1. $f_i = S[|f_1 \dots f_{i-1}| + 1]$, jos merkki $S[|f_1 \dots f_{i-1}| + 1]$ ei esiinny merkkijonossa $f_1 \dots f_{i-1}$,
2. muuten f_i on merkkijonon $f_i \dots f_z$ pisin alkuosa, joka on välillä $S[1..|f_1 \dots f_{i-1}|]$ alkava merkkijonon S osajono.

Esimerkiksi merkkijonon $S[1..10] = \texttt{abaabaabba}$ tekijöitä on viisi: $\texttt{a}, \texttt{b}, \texttt{a}, \texttt{abaab}$ ja $\texttt{ba}$. Minkä tahansa merkkijonon $S[1..n]$ tekijät voi määrittää käymällä S läpi vasemmalta oikealle. Edellisen tekijän päättyessä merkkiin $S[j-1]$ on seuraavan tekijän määrittämiseksi löydettävä sellainen $1 \leq i < j$, jolla loppuosien $S[i..n]$ ja $S[j..n]$ pisin yhteinen alkuosa on pisin. Siis on selvitettävä indeksi i ja pituus $\texttt{lce}_S(i, j)$. Tekijöihinjakoon on kehitetty runsaasti algoritmeja [6], ja niitä on optimoitu esimerkiksi tilavaativuutta, käytännön sovelluksia tai rinnakkaislaskentaa ajatellen. Algoritmien määrä on seurausta tekijöihinjaon hyödyllisyydestä merkkijonojen tiivistyksessä, indeksoinnissa ja etsinnässä. Tekijöiden määrä toimii myös yhtenä tiivistyvyyden mittana.

LCE-ongelma kytkeytyy merkkijonon loppuosien järjestämiseen. Jos merkkijonon aakkosto on järjestetty, kahden loppuosan järjestyksen määrää niiden ensimmäinen eroava merkki. Järjestyksen avulla voi luoda monissa merkkijonoalgoritmeissa käytettävän loppuosataulukon [37] ja sitä täydentävän LCP-taulukon (*longest common prefix*). Toisinaan LCE-ongelma nimetäänkin LCP-ongelmaksi, mutta loppuosataulukot lienevät tyypillisempi asiayhteys käsitteelle pisin yhteinen alkuosa.

Sovelluksissa `lce`-kyselyt vievät usein merkittävästi aikaa suhteessa muuhun laskentaan. Toisaalta nopeita kyselyjä tukevat tietorakenteet vievät helposti liikaa tilaa pitkiä merkkijonoja käsiteltäessä. Kyselyjen toteutuksessa onkin tärkeää optimoida aika- ja tilavaativuus sopivasti.

LCE-ongelma on hiljattain yleistetty [7] juurellisiin puihin, joissa solmut on nimetty merkeillä ja peräkkäisten solmujen merkit muodostavat merkkijonoja. Olkoot T tällainen puu sekä v_1, v_2, w_1 ja w_2 sellaiset solmut, että w_1 on solmun v_1 ja w_2 solmun v_2 jälkeläinen. Puulle T voi tehdä seuraavia kyselyjä:

- $\texttt{lce}_T(v_1, w_1, v_2, w_2)$ palauttaa polkuparin $(v_1 \rightsquigarrow w_1, v_2 \rightsquigarrow w_2)$ polkujen pisimmän yhteisen alkuosan päätepisteet (polku–polku-LCE).
- $\texttt{lce}_T(v_1, w_1, v_2)$ palauttaa pisimmän polku–polku-LCE:n palauttaman vastauksen polkuparien $(v_1 \rightsquigarrow w_1, v_2 \rightsquigarrow x_2)$ joukolle, missä x_2 on solmun v_2 jälkeläinen (polku–puu-LCE).
- $\texttt{lce}_T(v_1, v_2)$ palauttaa pisimmän polku–polku-LCE:n palauttaman vastauksen polkuparien $(v_1 \rightsquigarrow x_1, v_2 \rightsquigarrow x_2)$ joukolle, missä x_1 on solmun v_1 jälkeläinen ja x_2 on solmun v_2 jälkeläinen (puu–puu-LCE).

Kyselyjen tekeminen puille osoittautunee hyödylliseksi joissakin sovelluksissa. LCE-ongelman kahden merkkijonon muunnelman voi ratkaista ilman katenointia luomalla yleistetyn loppuosapuun, jossa jokainen kummankin merkkijonon loppuosa löytyy polkuna juuresta lehteen. Tähän puuhun tehtävällä kyselyllä saadaan selville kahden loppuosan pisin yhteinen alkuosa. Sama pätee useamman merkkijonon joukolle. Jäljempänä kerrotusti myös tavallinen LCE-ongelma voidaan ratkaista loppuosapuun avulla, mutta yleistetty puu saattaa käytännössä viedä vähemmän tilaa, jos merkkijonot ovat kovin toisteisia.

Puihin yleistettyjä kyselyjä voi käyttää apuna tehtäessä hakuja XML-dokumenteista. Näiden rakenteen voi tulkita puuksi, jonka tutkimista haut edellyttävät. Kyselyjen avulla polkuja ja alipuita voi verrata ilman nimenomaista puussa liikkumista.

Puihin yleistettyä LCE-ongelmaa ei käsitellä tutkielmassa enempää.

2.2 Perinteiset ratkaisut

Ratkaisu LCE-ongelmaan ilman apurakenteita on verrata merkkijonon $S[1..n]$ loppuosia $S[i..n]$ ja $S[j..n]$ merkki kerrallaan, kunnes löytyy eroava merkki tai merkki-

jono päättyy. Algoritmi toimii vakiotilassa eikä vaadi esikäsittelyä. Huonona puolena ratkaisun aikavaativuus on $O(\texttt{lce}_S(i,j)) = O(n)$.

Algoritmi on silti monesti nopea käytännössä, sillä suurin osa pisimmistä yhteisistä alkuosista on lyhyitä [28]. Olkoot aakkosto $[1..\sigma]$ ja aakkoston merkit riippumattomia ja samoin jakautuneita. Jos $\sigma \geq 2$ ja $n \geq 2$, niin joukolle, johon kuuluvat kaikki merkkijonot pituudeltaan n, on osoitettu pätevän, että keskimäärin pisin yhteinen alkuosa on lyhyempi kuin $\frac{1}{\sigma-1}$. Toisaalta maksimipituus voi olla huomattavasti tätä suurempi esimerkiksi luonnollisten kielten merkkijonoissa tai DNA-juosteissa, joissa merkkien jakaumaa koskeva oletus ei päde. Pahimman tapauksen aikavaativuuden pienentämiseksi on turvauduttava muihin algoritmeihin.

Tarkastellaan LCE-ongelman kahta tietorakenteita käyttävää ratkaisua, joista ensimmäinen perustuu loppuosataulukkoon. Jos aakkoston merkeillä on järjestys, merkkijonon $S[1..n]$ loppuosista voi luoda loppuosataulukon [37]. Se saadaan järjestämällä merkkijonon S loppuosat aakkosjärjestykseen ja tallentamalla taulukkoon $SA[1..n]$ loppuosien ensimmäisten merkkien indeksit. Näin ollen loppuosa $S[SA[i-1]..n]$ on aakkosjärjestyksessä ennen loppuosaa $S[SA[i]..n]$ kaikilla $1 < i \leq n$. LCE-ongelman ratkaisemiseen tarvitaan käänteinen loppuosataulukko $SA^{-1}[1..n]$. Sen arvo $SA^{-1}[i]$ kertoo, kuinka mones loppuosa $S[i..n]$ on järjestettyjen loppuosien joukossa. Siis $i = SA^{-1}[SA[i]]$. Kootaan vielä järjestyksessä vierekkäisten loppuosien pisimpien yhteisten alkuosien pituudet taulukkoon $LCP[1..n]$. Siinä arvo $LCP[i]$ on järjestyksessä loppuosan i ja sitä edeltävän loppuosan pisimmän yhteisen alkuosan pituus. Lisäksi $LCP[1] = 0$.

Määritellään lisäksi osavälin minimi -kysely (*range minimum query*), jota tarvitaan myöhemminkin. Se palauttaa argumenteilla $1 \leq i \leq j \leq n$ taulukon osavälin $A[i..j]$ pienimmän alkion indeksin, eli

$$\texttt{rmq}_A(i,j) = \arg\min_{i \leq k \leq j} A[k].$$

Jos osavälillä on useampi kuin yksi pienin alkio, kysely voi palauttaa niistä minkä tahansa indeksin, ellei erikseen mainita toisin.

Pisimmän yhteisen alkuosan selvittäminen perustuu havaintoon, että järjestetyssä loppuosien joukossa loppuosien $S[i..n]$ ja $S[j..n]$ pisin yhteinen alkuosa on samalla kaikkien näiden väliin jäävien loppuosien pisin yhteinen alkuosa. Järjestetyille loppuosille pätee nimittäin yleisesti, että loppuosan $S[i..n]$ ja sen viereisen loppuosan pisin yhteinen alkuosa on pitempi tai yhtä pitkä kuin loppuosan $S[i..n]$ ja sitä samassa suunnassa kauempana olevan loppuosan pisin yhteinen alkuosa. Tämän seurauksena kaikki loppuosien $S[i..n]$ ja $S[j..n]$ väliin jäävät loppuosat alkavat näiden kahden loppuosan pisimmällä yhteisellä alkuosalla.

Loppuosien $S[i..n]$ ja $S[j..n]$ paikat loppuosien järjestetyssä joukossa saadaan taulukosta SA^{-1}. Järjestetyn joukon kahden vierekkäisen loppuosan pisimmän yhteisen alkuosan pituus on LCP-taulukossa. Siis osavälin $LCP[SA^{-1}[i]+1..SA^{-1}[j]]$ arvoista pienin on loppuosien $S[i..n]$ ja $S[j..n]$ pisimmän yhteisen alkuosan pituus.

Nyt kysely $\texttt{rmq}_{LCP}(SA^{-1}[i]+1, SA^{-1}[j])$ palauttaa loppuosien $S[i..n]$ ja $S[j..n]$ pi-

Taulukko 1: Merkkijonon $S[1..6]$ = `ananas` loppuosataulukko SA, käänteinen loppuosataulukko SA^{-1}, pisimmät yhteiset alkuosat -taulukko LCP ja loppuosat. Esimerkiksi loppuosien $S[1..6]$ = `ananas` ja $S[4..6]$ = `nas` pisimmän yhteisen alkuosan pituus saadaan kyselyllä $\texttt{lce}_S(1,4) = \texttt{rmq}_{LCP}(SA^{-1}[1]+1, SA^{-1}[4]) = \texttt{rmq}_{LCP}(2,5) = \min\{3,1,0,2\} = 0$.

i	$SA[i]$	$SA^{-1}[i]$	$LCP[i]$	$S[SA[i]..6]$
1	1	1	0	`ananas`
2	3	4	3	`anas`
3	5	2	1	`as`
4	2	5	0	`nanas`
5	4	3	2	`nas`
6	6	6	0	`s`

simmän yhteisen alkuosan pituuden sisältävän alkion indeksin k. Sen avulla saadaan $\texttt{lce}_S(i,j) = LCP[k]$.

Taulukossa 1 on esimerkki edellä mainituista tietorakenteista ja $\texttt{lce}_S$-kyselystä.

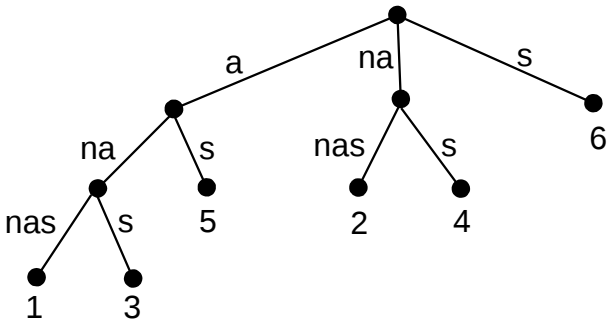
Kyselyn $\texttt{lce}_S$ aikavaativuus riippuu osavälikyselyn nopeudesta. Käytäessä läpi jokainen osavälin alkio aikavaativuus on $O(n)$, koska osavälin pituus on enintään $n-1$. Toinen triviaali tapa on taulukoida kaikki osavälikyselyjen vastaukset, jolloin vastaus saadaan taulukosta vakioajassa mutta tilavaativuudeksi tulee $O(n^2)$ konesanaa. Muitakin vaihtoehtoja on, niiden joukossa apurakennetta käyttävä vakioaikainen ratkaisu tilavaativuudeltaan $O(n)$ konesanaa [3]. Apurakenne voidaan luoda ajassa $O(n)$.

Taulukoiden SA, SA^{-1} ja LCP tilavaativuus on $O(n)$ konesanaa. Taulukot SA ja SA^{-1} ovat kokonaislukujen $1, 2, \ldots, n$ permutaatioita. LCP-taulukon alkioiden summa on suurin merkkijonon S sisältäessä vain yhtä merkkiä, jolloin taulukko sisältää alkiot $0, 1, \ldots, n-1$. Taulukoiden koko ei riipu merkkijonon S aakkostosta. Kunkin taulukon koko on esimerkiksi $4n$ tavua, jos kokonaisluku vie toteutuksessa tilaa neljä tavua.

Taulukot SA, SA^{-1} ja LCP voi luoda vakio- ja kokonaislukuaakkoston merkkijonosta ajassa $O(n)$ ja muulloin ajassa $O(n \log n)$ [16]. Pitkiä tekstejä varten on algoritmeja, joissa hyödynnetään rinnakkaisuutta tai minimoidaan ulkoisen muistin käyttöä [30].

Toinen tietorakenteita käyttävä LCE-ongelman ratkaisu perustuu loppuosapuuhun. Merkkijonosta $S[1..n]$ luotu loppuosapuu [54] on juurellinen puu, joka täyttää seuraavat ehdot:

1. Puussa on n lehteä numeroituna kokonaisluvuilla 1–n.
2. Jokainen kaari on merkitty vähintään yhden pituisella merkkijonon S osajonolla. Juuresta lehteen i vievän polun osajonojen katenaatio on täsmälleen loppuosa $S[i..n]$.



Kuva 1: Merkkijonon $S[1..6] =$ `ananas` loppuosapuu. Loppuosaa $S[i..6]$ vastaa polku juuresta lehteen i. Loppuosien $S[i..6]$ ja $S[j..6]$ pisintä yhteistä alkuosaa vastaa polku juuresta lehtien i ja j alimpaan yhteiseen esi-isään. Esimerkiksi loppuosilla $S[3..6] =$ `anas` ja $S[5..6] =$ `as` on yhteinen polku täsmälleen alkuosan `a` osalta.

3. Mille tahansa loppuosille $S[i..n]$ ja $S[j..n]$ pätee, että niiden polut juuresta lehtiin ovat sama polku täsmälleen loppuosien pisimmän yhteisen alkuosan osalta.

Kun aakkoston merkeillä on järjestys, on tavanomaista esittää puun polut järjestyksessä siten, että vasemmanpuoleisin polku vastaa järjestyksessä ensimmäistä loppuosaa. Ennen puun luomista merkkijonon S loppuun lisätään usein merkkijonossa esiintymätön merkki \$, joka varmistaa jokaisen loppuosan päättyvän puussa lehtisolmuun. LCE-ongelman ratkaisemista varten jokaiseen solmuun voi tallentaa osajonon pituuden juuresta kyseiseen solmuun.

Koska kahta loppuosaa vastaavat polut ovat sama polku täsmälleen loppuosien pisintä yhteistä alkuosaa vastaavalta osalta eli juuresta johonkin solmuun v, pisin yhteinen alkuosa on kaarten merkkien katenaatio juuresta solmuun v. Tällöin v on kyseisten loppuosien lehtisolmujen alin yhteinen esi-isä. Näin ollen LCE-ongelma palautuu loppuosien lehtisolmujen alimman yhteisen esi-isän etsinnäksi loppuosapuusta.

Kuvassa 1 on esimerkki loppuosapuusta ja alimmasta yhteisestä esi-isästä.

Loppuosapuuta käytettäessä LCE-ongelman aikavaativuus riippuu alimman yhteisen esi-isän etsinnän nopeudesta. Ilman apurakenteita esi-isän etsintä edellyttää $O(n)$ solmun läpikäyntiä. Esi-isän etsintään on kuitenkin olemassa [3] apurakennetta käyttävä vakioaikainen ratkaisu tilavaativuudeltaan $O(n)$ konesanaa. Apurakenteen luominen vie aikaa $O(n)$. Alimman yhteisen esi-isän ja taulukon osavälin minimin etsinnän on lisäksi osoitettu olevan yhteydessä toisiinsa, sillä edellinen ongelma voidaan palauttaa jälkimmäiseksi ja päinvastoin.

Loppuosapuun tilavaativuus on $O(n)$ konesanaa olettaen, että merkkijonon aakkosto sallii solmun toteuttamisen vakiotilassa [2]. Puussa on enintään $2n$ solmua, koska puussa on n lehteä ja tavanomaisessa toteutuksessa jokaisella sisäsolmulla vähintään kaksi lasta. Kaarten osajonojen tilavaativuus on $O(n^2)$ konesanaa, koska merkkijono sisältää enintään neliöllisen määrän osajonoja. Mutta osajonot voi korvata puussa

Taulukko 2: Kyselyn $\texttt{lce}_S(i,j)$ ilman tilaa säästäviä tietorakenteita toteutettujen ratkaisujen aika- ja tilavaativuudet. $S[1..n]$ on järjestetyn aakkoston merkkijono, ja tilavaativuudet on ilmaistu konesanoina. Rakenteitta toimiva ratkaisu ei tarvitse esikäsittelyä. Loppuosataulukkoon perustuva ratkaisu käyttää apurakennetta vakioaikaiseen osavälikyselyyn $\texttt{rmq}_{LCP}$. Loppuosapuuta täydennetään apurakenteella alimman yhteisen esi-isän (*lowest common ancestor*) vakioajassa palauttavaa kyselyä $\texttt{lca}_T$ varten. Loppuosapuun ja -taulukon luominen ajassa $O(n)$ edellyttää vakiotai kokonaislukuaakkostoa. Huomataan kyselyn $\texttt{lce}_S(i,j)$ aika–tila-vaihtokauppa: vastaus saadaan vakiotilassa ajassa $O(n)$ tai vakioajassa tilassa $O(n)$ konesanaa.

	Käyttö		Esikäsittely	
	Aikav.	Tilav.	Aikav.	Tilav.
Ei rakenteita	$O(n)$	$O(1)$	-	-
SA^{-1}, LCP, $\texttt{rmq}_{LCP}$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Loppuosapuu T, $\texttt{lca}_T$	$O(1)$	$O(n)$	$O(n)$	$O(n)$

kokonaislukupareilla, jotka viittaavat osajonon alku- ja loppukohtaan alkuperäisessä merkkijonossa. Tällöin kaarten tilavaativuus pienenee lineaariseksi.

Lapsisolmuihin viittaavien linkkien toteutus vaikuttaa merkittävästi loppuosapuun viemään tilaan ja käytön nopeuteen [23]. Keskeisiä tapoja ovat taulukko, linkitetty lista, tasapainotettu puu ja hajautustaulu. Eniten lapsia on yleensä solmuilla lähellä puun juurta, jolloin taulukko voi olla hyvä valinta mahdollistaen vakioaikaisen haun. Alempana puussa lasten määrän vähentyessä linkitetty lista voi olla parempi vaihtoehto taulukon tilantarpeen vuoksi. Optimoimaton loppuosapuu voi viedä tilaa 20 tavua merkkijonon merkkiä kohti mutta optimoitu puu esimerkiksi puolet tästä.

Loppuosapuun voi luoda ajassa $O(n)$, jos aakkosto on vakioaakkosto [51] tai kokonaislukuaakkosto [14]. Muuten luominen vie aikaa $O(n \log n)$.

Taulukkoon 2 on koottu luvussa esitettyjen LCE-ongelman ratkaisujen aika- ja tilavaativuudet. Ratkaisuissa ei hyödynnetä tilaa säästäviä tietorakenteita, vaikka rakenteiden sisältämä informaatio mahtuisi pienempäänkin tilaan. Käytännössä tietorakenteet vievät tilaa moninkertaisesti alkuperäiseen merkkijonoon verrattuna, jos aakkoston merkki mahtuu esimerkiksi yhteen tavuun. Sovellukset ovat monesti liian hitaita, jos tietorakenne ei mahdu keskusmuistiin. Etenkin perinteisen puun kohdalla hitautta lisää se, ettei muistin lokaalisuutta hyödynnetä tehokkaasti.

Tietorakenteiden luomisen jälkeen ei tarvita alkuperäistä merkkijonoa S, koska kysely $\texttt{lce}_S(i,j)$ palauttaa alkuosan pituuden itse alkuosan sijasta. Loppuosataulukkoa käytettäessä taulukoiden SA^{-1} ja LCP tiedot riittävät kyselyyn vastaamiseen. Loppuosapuussa riittää kuhunkin solmuun tallennettu tieto juuresta solmuun vievää polkua vastaavan osajonon pituudesta. Alkuperäisen merkkijonon hävittäminen ei kuitenkaan paranna ratkaisujen tilavaativuusluokkia.

Tilaa säästäviksi ratkaisuiksi on kehitetty tiivistettyjä loppuosapuita [21]. Tilan minimoinnista huolimatta esimerkiksi etsintä puusta on samassa aikavaativuusluokas-

sa kuin tiivistämättömästä puusta. Käytännössä loppuosataulukon käyttö on silti usein nopeampaa. Puun toiminnallisuuden säilyttävä taulukko apurakenteineen vie tilaa kuusi tavua alkuperäisen merkkijonon merkkiä kohti [1]. Seuraavassa luvussa paneudutaan tarkemmin tiedon tilavaativuuteen.

3 Tiedon tilavaativuus

Luvun alussa käsitellään tiedon tiivistyvyyttä yksittäisiin tiivistysalgoritmeihin syventymättä. Esitellään entropian merkitys tilavaativuuden alarajana ja yhteys entropiakoodaukseen. Entropian avulla tarkastellaan tilaa säästävien tietorakenteiden tilantarvetta. Alkuperäisen informaation säilyttävien rakenteiden jälkeen tutustutaan koodaavien tietorakenteiden luomiseen ja käyttöön.

3.1 Tiivistys ja entropia

Suuren tietomäärän hallintaa voi helpottaa tiivistämällä eli korvaamalla tieto alkuperäistä lyhyemmällä kuvauksella. Menetelmiä on useita, ja ne voidaan jakaa häviöttömiin ja häviöllisiin.

Häviöttömässä tiivistyksessä alkuperäinen tieto voidaan palauttaa purkamalla, joten tiedon informaatiosisältö ei vähene tiivistettäessä. Jos tiivistetty esitys vie vähemmän tilaa kuin alkuperäinen, tiedon tilastollinen redundanssi on vähentynyt. Muun muassa erilaisia tekstejä tiivistetään häviöttömästi, ja tärkeimpiin tapoihin kuuluvat juoksupituuskoodaus, kielioppimenetelmät ja sanakirjamenetelmät. Tiivistäminen Lempel–Ziv-tekijöihinjaon avulla on esimerkki viimeksi mainitusta, sillä tekijät muodostavat sanakirjan, josta alkuperäisen tiedon voi palauttaa.

Häviöllisissä menetelmissä alkuperäistä tietoa ei voi täysin palauttaa tiivistetystä esityksestä, eli tilastollisen redundanssin lisäksi tiedon informaatiosisältö vähenee. Tyypillisesti ääniä, kuvia, videoita ja tietokonegrafiikkaa tiivistetään häviöllisesti, jolloin niistä poistetaan havaintokyvyn ulottumattomiin jääviä vivahteita tai muita käyttötarkoitukseen tarpeettomia osia. Mitä pienempään tilaan tieto tiivistetään häviöllisesti, sitä enemmän informaatiota menetetään.

Tiivistysmenetelmästä riippumatta keskeinen kompromissi on valinta tiivistys- ja purkamisnopeuden sekä tiivistyssuhteen välillä. Tiivistyssuhteella tarkoitetaan tiivistetyn tiedon ja alkuperäisen tiedon koon suhdetta. Mitä nopeampi algoritmi valitaan, sitä vähemmän tieto yleensä tiivistyy. Sopivan algoritmin valinta riippuu tilanteesta: esimerkiksi reaaliaikaisessa tiivistyksessä korostuu algoritmin nopeus, kun taas säilytystilaa minimoitaessa pyritään mahdollisimman hyvään tiivistyssuhteeseen.

Informaatioteoreettisella entropialla tarkoitetaan karkeasti ottaen jonkin joukon informaatiosisältöä. Näin ollen entropian avulla saadaan laskettua joukon alkioiden tilavaativuuden alaraja. Tietoa tiivistettäessä pyritään lähestymään alarajaa, mut-

ta tietoa ei voida häviöttömästi tiivistää sitä pienempään tilaan. Tarkastellaan esimerkkeinä pahimman tapauksen entropiaa ja Shannon-entropiaa.

Pahimman tapauksen entropia [40] kertoo bittien vähimmäismäärän, joka tarvitaan yksilöimään joukon alkio. Joukon A pahimman tapauksen entropia on

$$H_p(A) = \log |A|.$$

Joukon alkion yksilöivää bittijonoa sanotaan koodiksi. Jos kaikki koodit ovat yhtä pitkiä ja kokonaisluvun pituisia, joukon A alkion koodin pituus on vähintään $\lceil \log |A| \rceil$. Koodien ollessa eripituisia pisimmät koodit ovat silti vähintään näin pitkiä, mistä tulee nimitys pahimman tapauksen entropia.

Pahimman tapauksen entropia kaikkien n bitin pituisten bittijonojen joukolle B on $H_p(B) = \log |B| = \log(2^n) = n$ bittiä. Tarvitaan siis n bittiä yksilöimään yksi joukon bittijono. Jos puolestaan kyseessä on kaikkien aakkostosta $[1..\sigma]$ muodostettujen n merkin pituisten merkkijonojen joukko S, sen pahimman tapauksen entropia on $H_p(S) = \log |S| = \log(\sigma^n) = n \log \sigma$ bittiä.

Entropiakoodaus on olennainen osa useimpia tiivistysmenetelmiä. Perusajatuksena on korvata tiedossa esiintyvät aakkoston merkit niitä keskimäärin lyhyemmillä vastineilla ja siten minimoida merkkien viemä tila. Merkkijonossa usein esiintyvät merkit kannattaa korvata lyhyemmillä vastineilla kuin harvoin esiintyvät. Tämä periaate voidaan formalisoida informaation [48] käsitteen avulla: merkki, jonka todennäköisyys on p, sisältää $\log(1/p)$ bittiä informaatiota. Mitä todennäköisempi merkki on, sitä vähemmän informaatiota se sisältää, ja päinvastoin. Tämä havainto muodostaa yhteyden merkkijonon merkkien todennäköisyysjakauman ja merkkijonon informaatiosisällön välille.

Merkkien todennäköisyysjakaumasta voidaan laskea keskimääräinen bittien vähimmäismäärä, joka tarvitaan merkkiä kohti. Tämä Shannon-entropia lasketaan merkkijonolle T kaavalla

$$H_s(T) = \sum_{t=1}^{\sigma} p_t \log(1/p_t) = -\sum_{t=1}^{\sigma} p_t \log p_t,$$

missä σ on aakkoston merkkien määrä ja p_t merkin t todennäköisyys merkkijonossa. Jos jokainen merkki on yhtä todennäköinen, Shannon-entropia on suurimmillaan ja vastaa joukon pahimman tapauksen entropiaa. Esimerkiksi aakkoston $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ merkkijonolle $T[1..6] = \mathtt{aababa}$ pätee $p_{\mathtt{a}} = \frac{4}{6}$ ja $p_{\mathtt{b}} = \frac{2}{6}$. Näin ollen $H_s(T) = -(\frac{4}{6} \log \frac{4}{6} + \frac{2}{6} \log \frac{2}{6}) \approx 0{,}918$. Tämä on pienempi kuin aakkoston pahimman tapauksen entropia $H_p(\Sigma) = \log |\Sigma| = \log 2 = 1$.

Merkkijonon tilastollinen redundanssi voidaan määritellä tässä yhteydessä entropian avulla: redundanssi lasketaan vähentämällä merkkien koodien todellisten pituuksien todennäköisyyksillä painotetusta keskiarvosta merkkien Shannon-entropia. Käytännössä redundanssi ilmenee siten, että merkkien koodit ovat keskimäärin pitempiä kuin on välttämätöntä merkkien yksilöimiseksi.

Shannon-entropia ilmaisee yhden teoreettisen alarajan tiedon tiivistyvyydelle. Alaraja voi olla alempikin, jos tiivistettäessä käytetään hyväksi todennäköisyysjakauman lisäksi muita merkkijonon ominaisuuksia, esimerkiksi merkkien kontekstia tai osajonojen toisteisuutta. Joka tapauksessa entropiakoodauksella pyritään vähentämään merkkijonon redundanssia ja siten lähestymään jollakin tavalla laskettua entropiaa.

Huffman-koodaus [26] on häviöttömässä tiivistyksessä yleisesti käytettävä entropiakoodauksen muoto. Koodaus on alkuosakoodi, jossa jokainen aakkoston merkki koodataan bittijonoksi siten, ettei koodi ole minkään toisen koodin alkuosa. Todennäköisemmät merkit saavat lyhyemmät koodit kuin harvemmin esiintyvät.

Huffman-koodauksen vahvuuksia ovat nopea koodaus ja purkaminen. Heikkoutena koodien keskimääräinen pituus on yleensä suurempi kuin Shannon-entropia, sillä algoritmissa merkkien todennäköisyydet pyöristetään implisiittisesti luvun 1/2 potensseiksi. Lisäksi jokainen merkkijonon merkki koodataan erikseen. Jokainen merkki vaatii siten vähintään yhden bitin, vaikka esimerkiksi todennäköisyydellä 9/10 esiintyvä merkki sisältää vain $\log(10/9) \approx 0{,}152$ bittiä informaatiota. Huffman-koodauksen heikkoudet korostuvat, jos aakkoston koko on pieni tai yhden merkin todennäköisyys suuri.

Huffman-koodauksen on kuitenkin osoitettu olevan alkuosakoodiksi optimaalinen. Kun merkkijonon todennäköisyysjakauma tunnetaan, koodien keskimääräisen pituuden yläraja merkkijonolle $T[1..n]$ on $H_s(T)+1$. Siten merkkijono vie koodattuna tilaa vähemmän kuin $n(H_s(T)+1)$ bittiä.

Aritmeettisella koodauksella [25] päästään Huffman-koodausta lähemmäs entropiaa. Merkkijono $T[1..n]$ vie koodattuna tilaa vähemmän kuin $n(H_s(T)) + 2$ bittiä eli vähemmän kuin kaksi lisäbittiä koko merkkijonoa kohti. Koodauksessa voidaan käyttää lähes tarkkaa merkkien todennäköisyysjakaumaa, eikä merkkejä korvata koodeilla yksitellen. Sen sijaan koko merkkijono koodataan yhdeksi luvuksi x, joka on teoriassa mielivaltaisen tarkka liukuluku välillä $0{,}0 \leq x < 1{,}0$. Koodauksen aikana informaatio esitetään kahden liukuluvun välinä.

Aritmeettinen koodaus voidaan tulkita Huffman-koodauksen yleistykseksi siinä mielessä, että kumpikin menetelmä tuottaa pituudeltaan saman lopputuloksen, jos jokaisen merkin todennäköisyys on luvun 1/2 potenssi. Aritmeettinen koodaus on kuitenkin Huffman-koodausta hitaampaa algoritmissa käytettävien aritmeettisten operaatioiden takia.

Epäsymmetriset lukujärjestelmät [13] ovat viime vuosina kehitettyjä entropiakoodauksen muotoja. Niissä yhdistyvät Huffman-koodauksen ja aritmeettisen koodauksen hyvät puolet eli nopeus ja lähes tarkan todennäköisyysjakauman käyttö. Koodauksen aikana informaatio esitetään yhtenä luonnollisena lukuna x siten, että x sisältää kulloinkin noin $\log x$ bittiä informaatiota.

Useimpia tiivistysmenetelmiä käytettäessä tiivistetty tieto tukee suoraan vain yhtä operaatiota, purkamista. Tiivistetty tieto on purettava ennen muokkaamista tai hakujen tekemistä. Sama pätee entropiakoodattuun tietoon, jos menetelmänä käyte-

tään aritmeettista koodausta tai epäsymmetrisiä lukujärjestelmiä. Koodauksen tuloksena saadusta luvusta ei voida suoraan lukea alkuperäistä tietoa. Tiivistyksen hyötynä on tällöin tallennustilan säästäminen tai tietoverkossa välitettävän tiedon koon minimointi.

Sen sijaan Huffman-koodattu tieto voidaan lukea ilman purkamista, jos koodit tunnetaan. Koodauksessa jokainen yksittäinen merkki vain korvataan yksikäsitteisellä vastineellaan, joten tieto säilyy rakenteeltaan suoraan luettavana. Huffman-koodatun tiedon pituus onkin käyttökelpoinen mitta arvioitaessa tilaa säästävien tietorakenteiden tilantarvetta.

Tilaa säästäviä tietorakenteita on kehitetty erityisesti suurten tietomäärien käsittelyyn korvaamaan perinteisiä, enemmän tilaa vieviä rakenteita. Tilaa säästävien tietorakenteiden yhteydessä oletetaan yleensä, niin myös tässä tutkielmassa, ettei tietorakenteita tarvitse purkaa ennen haluttujen operaatioiden suoritusta. Tietorakenteet eroavat näin perinteisten tiivistysalgoritmien tuloksista.

3.2 Tietorakenteiden luokittelu

Tietorakenteen avulla järjestetään ja tallennetaan tietoa siten, että rakenne tukee kulloinkin tarvittavia operaatioita aika- ja tilavaativuuden kannalta tehokkaasti. Tehokkuuden arvioinnissa on kolme keskeistä tekijää: tietorakenteen tilavaativuus, operaatioiden suorituksen aikavaativuus ja tietorakenteen luomisen eli esikäsittelyn aika- ja tilavaativuus. Ihanteellisessa tapauksessa tietorakenne on mahdollisimman pieni, halutut operaatiot vakioaikaisia ja esikäsittely nopeaa ja tilankäytöltään tehokasta. Suuria tietomääriä käsiteltäessä sopiva ratkaisu on useimmiten kompromissi.

Jäljempänä käytetään pieni o -notaatiota [40]. Se tarkoittaa ylärajaa, jota hitaammin algoritmin aika- tai tilavaativuus kasvaa asymptoottisesti: kahdelle funktiolle pätee $g(n) \in o(f(n))$, jos raja-arvo $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$. Jos tietorakenteen tilavaativuus on $2n + o(n)$ bittiä, rakenteeseen tarvitaan $2n$ bittiä ja alilineaarisesti kasvava määrä bittejä, esimerkiksi yhteensä $2n + O(\frac{n}{\log n})$ bittiä. Jos n bittiä käsitellään vakioajan vievissä, konesanan kokoisissa osissa w, saadaan aikavaativuudeksi $O(\frac{n}{w}) = O(\frac{n}{\log n}) = o(n)$. Merkintä $o(1)$ tarkoittaa, että funktion arvo lähestyy nollaa, kun n kasvaa rajatta, esimerkiksi $\frac{\log\log n}{\log n} = o(1)$.

Tilavaativuuden kannalta tietorakenteet voi jakaa perinteisiin ja tilaa säästäviin. Perinteisten tietorakenteiden tilavaativuutta pidetään optimaalisena, jos se on lineaarinen suhteessa syötteen kokoon. Jos syöte vie tilaa n konesanaa, tietorakenteen optimaalinen koko on $O(n)$ konesanaa tai vastaavasti $O(n \log n)$ bittiä.

Tilaa säästävien tietorakenteiden nimitykset eivät ole täysin vakiintuneita. Kompakti tietorakenne (*compact data structure*) on yleisnimitys tietorakenteelle, joka vie vähemmän tilaa kuin vastaava perinteinen rakenne. Tiivis tietorakenne (*succinct data structure*) puolestaan vie lisätilaa alilineaarisesti alkuperäiseen tietoon verrattuna, esimerkiksi n bitin taulukkoon liittyvä rakenne taulukon kanssa yhteensä $n + o(n)$ bittiä. Tiivistetyn tietorakenteen (*compressed data structure*) tilavaativuus voi olla

pienempi kuin tiivistämättömän alkuperäisen tiedon. Tiivistettyä rakennetta saatetaan sanoa täysin tiivistetyksi (*fully compressed*), jos sen tilavaativuus on alkuperäisen tiedon kanssa yhteensä $H + o(H)$, missä H on jollakin mallilla laskettu alkuperäisen tiedon entropia.

Toinen tapa ryhmitellä tilaa säästävät tietorakenteet on tutkia, säilyttävätkö ne alkuperäisen tiedon. Olkoon alkuperäinen tieto kooltaan n bittiä. Systemaattinen tietorakenne (*systematic data structure*) vie tilaa alilineaarisesti alkuperäiseen tietoon nähden. Alkuperäinen tieto säilytetään ja on tarpeen rakenteen toiminnalle, joten yhteensä tilavaativuus on $n + o(n)$ bittiä. Ei-systemaattinen (*non-systematic*) tietorakenne vie myös tilaa alilineaarisesti, mutta alkuperäistä tietoa ei säilytetä, joten tilavaativuus on $o(n)$ bittiä. Ei-systemaattinen rakenne ei sisällä kaikkea alkuperäistä informaatiota, mutta sen avulla voidaan vastata ennalta määriteltyihin kyselyihin alkuperäisestä tiedosta.

Systemaattista tietorakennetta nimitetään kirjallisuudessa myös indeksoivaksi tietorakenteeksi (*indexing data structure*). Ei-systemaattisesta tietorakenteesta käytetään joskus nimityksiä koodaava tietorakenne (*encoding data structure* tai *encoding*) ja abstrakti tietotyyppi (*abstract data type*). Käsitettä koodaava tietorakenne voi perustella sillä, että ei-systemaattinen tietorakenne koodataan jollakin tavalla alkuperäisestä tiedosta. Abstrakti tietotyyppi puolestaan viittaa siihen, ettei tietorakenne sisällä kaikkea alkuperäistä tietoa sellaisenaan tai tiivistettynä.

Tutkielman aihe huomioon ottaen on keskeistä erottaa kaiken alkuperäisen informaation sisältävät tai tarvitsevat tietorakenteet muista rakenteista. Edellisiä nimitetään jatkossa perinteisiksi tai systemaattisiksi tietorakenteiksi ja jälkimmäisiä koodaaviksi tietorakenteiksi.

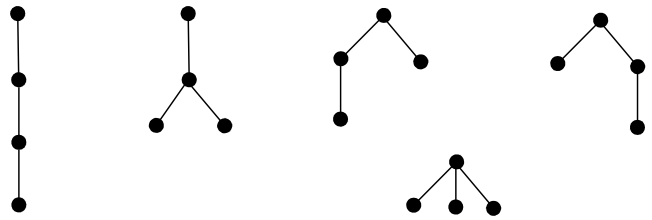
3.3 Taulukko ja puu

Tarkastellaan taulukon ja puun tilavaativuutta. Olkoon taulukko $A[1..n]$ peräkkäisistä alkioistaan koostuva tietorakenne. Jos alkiot ovat yksittäisiä bittejä eli A on bittivektori, A vie tilaa vähintään n bittiä. Aakkoston $[1..\sigma]$ alkioista koostuva A vie tilaa vähintään $n \log \sigma$ bittiä, koska jokainen aakkoston merkki vie tilaa $\log \sigma$ bittiä tullakseen yksilöidyksi. Konesanaan mahtuvien kokonaislukujen taulukko A vie tilaa $O(n \log n)$ bittiä olettaen konesanan kooksi $O(\log n)$ bittiä. Sopivalla entropiakoodauksella tilantarve voi laskea esimerkkejä alemmaksi.

Osoittimilla voi luoda tietorakenteessa yhteyksiä alkioiden välille, jolloin siirtyminen alkiosta toiseen on vakioaikaista. Osoittimet myös helpottavat ylläpitoa, jos tietorakenne on dynaaminen eli muuttuu käytön aikana.

Osoittimet lisäävät tilantarvetta, koska ne on tallennettava tietoalkioiden ohella. Osoitin joukon $\{1, 2, \ldots, n\}$ alkioon vie tilaa $\log n$ bittiä. Yleisesti $O(n)$ osoitinta sisältävä tietorakenne vie tilaa vähintään $O(n \log n)$ bittiä. Esimerkiksi perinteinen puu vie tilaa vähintään $n \log n$ bittiä, jos puussa on n solmua.

Verrataan perinteisen puun tilantarvetta puun pahimman tapauksen entropiaan.



Kuva 2: Kaikki viisi erilaista neljän solmun järjestyspuuta.

Järjestyspuussa (*ordinal tree*) [5] on juuri ja solmujen lapsilla järjestys (esimerkiksi vasemmalta oikealle). Lasten määrää ei rajoiteta. Kombinatoriikalla on selvitetty, että n solmua sisältävien järjestyspuiden määrä vastaa solmujen määrää edeltävää Catalanin lukua [40]. Kun n saa arvot $1, 2, 3, 4$ ja 5, puiden määrä $|T_n|$ on vastaavasti $1, 1, 2, 5$ ja 14. Kuvassa 2 näkyy esimerkkinä kaikki viisi erilaista neljän solmun puuta.

Neljän solmun järjestyspuiden pahimman tapauksen entropia on $H_p(T_4) = \log |T_4| = \log 5 \approx 2{,}32$. Puut voi siten yksilöidä koodeilla, joiden pituus on $\lceil \log 5 \rceil = 3$, vaikkapa koodeilla `000`, `001`, `010`, `011` ja `100`. Jos jokaisen puun esiintymistodennäköisyys on $1/5$, joukon T_4 Shannon-entropia on suurimmillaan eli $H_s(T_4) = 5 \cdot 1/5 \cdot \log 5 \approx 2{,}32$, joka on sama kuin pahimman tapauksen entropia. Tällöin puut yksilöivät kanoniset Huffman-koodit [47] ovat `00`, `01`, `10`, `110` ja `111`. Huffman-koodien keskipituudeksi saadaan $3 \cdot 1/5 \cdot 2 + 2 \cdot 1/5 \cdot 3 = 2{,}40$.

Järjestyspuiden määrä $|T_n|$ lasketaan [40] yleisesti kaavalla

$$|T_n| = \frac{1}{n}\binom{2n-2}{n-1} = \frac{(2n-2)!}{n!\,(n-1)!}.$$

Kertomaa voi arvioida Stirlingin kaavalla $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + O(\frac{1}{n}))$. Sijoitetaan se edelliseen kaavaan ja sievennetään, jolloin saadaan puiden määräksi

$$|T_n| = \frac{4^n}{n^{3/2}} \cdot \Theta(1).$$

Tästä voi laskea järjestyspuiden pahimman tapauksen entropiaksi

$$H_p(T_n) = \log |T_n| = \log\left(\frac{4^n}{n^{3/2}} \cdot \Theta(1)\right) = 2n - \Theta(\log n).$$

Näin ollen järjestyspuun, jossa on n solmua, yksilöimiseksi riittää aina $2n$ bittiä, vaikka perinteinen puu vie tilaa vähintään $n \log n$ bittiä. Kaavan perusteella $2n$ bittiä on asymptoottisesti optimaalinen tilavaativuus. Neljän solmun puiden esimerkissä tekijän $\Theta(\log n)$ vaikutus on merkittävä ja yksilöintiin riittävät lyhyemmät koodit. Mutta esimerkiksi Huffman-koodit eivät säilytä puiden rakennetta, jolloin liikkuminen solmusta toiseen ei ole enää mahdollista. Ratkaisuksi on kehitetty tilaa säästäviä puita [46], joiden tilavaativuus on $2n + o(n)$ bittiä. Ne tukevat useita operaatioita vakioajassa, joitakin operaatioita jopa tehokkaammin kuin perinteinen

puu. Yleinen tapa toteuttaa tilaa säästävä puu on tallentaa itse puu $2n + O(1)$ bittiä käyttäen ja haluttuja operaatioita vakioajassa tukeva aputietorakenne $o(n)$ bittiä käyttäen. Toinen toteutustapa on pitää kaikki tarvittava tieto samassa tietorakenteessa. On esitetty myös järjestyspuita tarkkarajaisempia tilavaativuustuloksia [12], kuten useita vakioaikaisia operaatioita tukevien binääripuiden toteuttaminen $2n + n/(\log n)^{\Theta(1)}$ bittiä käyttäen.

On kolme keskeistä tapaa esittää n solmua sisältävä puu hyödyllisiä operaatioita tukevalla tavalla noin $2n$ bittiä käyttäen [11]. Tasoittainen unaarinen koodaus (*level-order unary degree sequence, LOUDS*) kuvaa puun bittivektorina $B[1..2n + 1]$. Se alkaa teknisisistä syistä biteillä `10`, jonka jälkeen puu käydään juuresta alkaen läpi taso kerrallaan. Jokainen solmu v kuvataan taulukkoon bittijonona $\texttt{1}^c\texttt{0}$, missä $\texttt{1}^c$ on solmun v lasten määrän verran `1`-bittejä. Bittivektoriin tulee yhteensä $n + 1$ kappaletta `0`-bittejä ja n kappaletta `1`-bittejä. Tässä puuesityksessä helposti toteutettavien operaatioiden määrä on pienempi kuin kahdessa muussa.

Toinen tapa esittää puu on syvyyssuuntainen unaarinen koodaus (*depth-first unary degree sequence, DFUDS*). Bittivektori $B[1..2n + 2]$ alkaa teknisisistä syistä biteillä `110`, jonka jälkeen puu käydään juuresta alkaen läpi esijärjestyksessä. Jokaisen uuden solmun kohdalla bittivektoriin lisätään solmun kuvaus $\texttt{1}^c\texttt{0}$. Koodauksessa jokaisen solmun alipuu kuvautuu bittivektoriin yhtäjaksoiselle osavälille. Esimerkiksi lapseen siirtyminen on helppo toteuttaa tässä esityksessä vakioajassa.

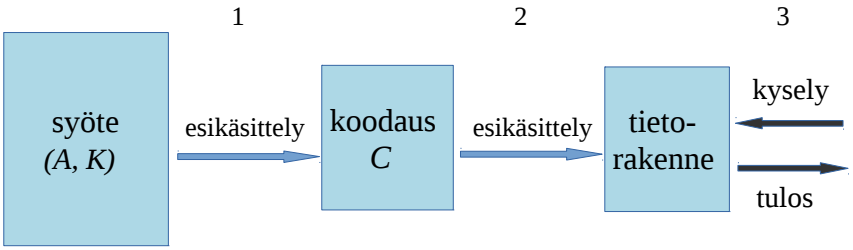
Kolmanneksi puun voi esittää bittivektorina $B[1..2n]$ käymällä puu läpi syvyysjärjestyksessä. Bittivektoriin B lisätään `1` saavuttaessa solmuun ensimmäisen kerran ja `0` poistuttaessa solmusta viimeisen kerran. Esimerkiksi kuvan 2 kahden vasemmanpuoleisimman puun bittivektorit ovat `11110000` ja `11101000`. Jos merkit `1` ja `0` tulkitaan sulkeiksi `(` ja `)`, bittivektorit vastaavat puun esittämistä tasapainoisten sulkeiden (*balanced parentheses*) avulla eli jonoina `(((())))` ja `(((())))`. Esityksen avulla voi toteuttaa runsaasti operaatioita, ja niille on olemassa myös nopea käytännön toteutus.

Tilaa säästäviä tietorakenteita käytettäessä on olennaista säilyttää sopiva tasapaino rakenteiden viemän tilan ja haluttujen operaatioiden aikavaativuuden välillä. Entropian avulla saadaan laskettua tietorakenteen vähimmäiskoko, joka voidaan tulkita rakenteen optimaaliseksi kooksi.

3.4 Koodaava tietorakenne

Koodaava tietorakenne [45] on tietorakenne, jota käytettäessä alkuperäiseen tietoon päästään käsiksi vain tietorakenteen kautta ennalta määriteltyjen kyselyjen avulla. Alkuperäistä tietoa ei tarvita tietorakenteen luomisen jälkeen. Tyypillisesti koodaava tietorakenne on kooltaan pienempi, jopa alilineaarinen alkuperäiseen tietoon verrattuna ja sisältää tätä vähemmän informaatiota. Tällöin alkuperäistä tietoa ei voi palauttaa tietorakenteesta.

Toisinaan kirjallisuudessa nimitetään koodaavaksi myös alkuperäisen tiedon säilyt-



Kuva 3: Koodaavan tietorakenteen luominen ja käyttö [45].

tävää, kooltaan tätä pienempää tietorakennetta. Tällaisen voi luoda, jos alkuperäistä tietoa voi tiivistää redundanssin vuoksi. Jos tietorakenne sisältää kaiken alkuperäisen informaation ja alkuperäisen tiedon voi kenties palauttaakin kyselyjen avulla, tietorakenne ei ole lainkaan koodaava, tai sitä voi pitää triviaalina koodaavana tietorakenteena.

Koodaava tietorakenne on käyttökelpoinen, jos ennalta määriteltyihin kyselyihin vastaamiseen ei tarvita kaikkea alkuperäistä informaatiota ja tietorakenteesta halutaan mahdollisimman pieni. Näin tarpeellista tietoa mahtuu keskusmuistiin ja nopeisiin välimuisteihin enemmän kuin systemaattisia tietorakenteita käytettäessä.

Määritellään yleinen malli koodaavan tietorakenteen luomiseen ja käyttöön. Syötteenä annetaan pari (A, K), missä A on tietoalkioiden joukko ja K haluttujen kyselyjen joukko. Joukko A voi sisältää esimerkiksi kaikki kokonaislukujen jonkin osajoukon alkioista muodostetut taulukot ja K osavälikyselyjä. Määritellään alkioille $a, b \in A$ relaatio $\sim$ seuraavasti: $a \sim b$, jos kaikki joukon K kyselyt palauttavat alkioilla a ja b saman vastauksen. Relaatio $\sim$ on refleksiivinen, symmetrinen ja transitiivinen ja näin ollen ekvivalenssirelaatio. Relaatio siis osittaa joukon A ekvivalenssiluokkiin siten, että jokainen joukon alkio kuuluu täsmälleen yhteen luokkaan. Mikä tahansa ekvivalenssiluokan alkio voidaan valita luokkansa edustajaksi y, joka riittää vastaamaan kyselyihin kaikkien samaan luokkaan kuuluvien alkioiden osalta.

Koodaavan tietorakenteen luominen ja käyttö voidaan jakaa kolmeen vaiheeseen kuvan 3 mukaisesti:

1. Joukko A ositetaan relaation $\sim$ mukaisiin ekvivalenssiluokkiin. Jokaisella ekvivalenssiluokalla on edustaja, jota kutsutaan luokkansa alkioiden koodaukseksi. Koodaukset muodostavat joukon C.

2. Koodauksista luodaan koodaava tietorakenne. Koodaukset sisältävät riittävästi informaatiota joukon K kyselyihin vastaamiseksi, ja myös tietorakenteella on tämä ominaisuus.

3. Tietorakennetta käytetään tekemällä joukkoon K kuuluvia kyselyjä.

Vaiheet 1 ja 2 ovat esikäsittelyä, joka tehdään kerran. Kun joukko C on luotu vaiheessa 1, joukko A jää tarpeettomaksi. Tietorakenne luodaan vaiheessa 2 käyttäen vain joukkoa C. Joukkoa A ei käytetä myöskään vaiheessa 3 kyselyjä tehtäessä, vaan kyselyt ovat ainoa tapa päästä käsiksi alkuperäiseen tietoon.

Koodaavien tietorakenteiden tilavaativuuden arvioimiseksi on hyödyllistä määritellä efektiivinen entropia [22]. Joukon A efektiivinen entropia joukon K suhteen (tai lyhyemmin joukon K efektiivinen entropia) bitteinä on

$$H_e(A) = \log |C|.$$

Toisin sanoen joukon A efektiivinen entropia joukon K suhteen vastaa joukon C pahimman tapauksen entropiaa. Nyt arvo $\lceil H_e(A) \rceil = \lceil \log |C| \rceil$ kertoo, monenko bitin koodi tarvitaan yksilöimään jokainen joukon C alkio, jotta joukon K kaikkiin kyselyihin voidaan vastata oikein. Jos käytettävät koodit ovat tätä lyhyempiä, pätee $2^{H_e(A)} < |C|$, jolloin vähintään kaksi joukon C alkiota saa saman koodin.

Efektiivinen entropia selittää koodaavan tietorakenteen tilavaativuuden informaatioteoreettisesti. Alkion $x \in A$ voi tietorakennetta luotaessa korvata alkion ekvivalenssiluokan edustajalla $y \in C$. Näin tietorakenteen ideaalinen koko on $(1 + o(1)) \log |C|$ bittiä. Alkuperäistä tietoa pienemmän koon saavuttamiseksi täytyy päteä $|C| < |A|$ ja alilineaarisen koon saavuttamiseksi $|C| = o(|A|)$. Jos puolestaan C sisältää yhtä paljon informaatiota kuin A, pätee $|C| = |A|$.

Koodaaville tietorakenteille voidaan määritellä myös odotettu efektiivinen entropia [45]. Se ottaa huomioon ekvivalenssiluokkien koot ja vastaa analogisesti Shannon-entropiaa. Joukon A odotettu efektiivinen entropia bitteinä joukon K suhteen lasketaan kaavalla

$$H_o(A) = \sum_{y \in C} \frac{|[y]_\sim|}{|A|} \log \frac{|A|}{|[y]_\sim|} = -\sum_{y \in C} \frac{|[y]_\sim|}{|A|} \log \frac{|[y]_\sim|}{|A|},$$

missä $[y]_\sim$ on alkion y ekvivalenssiluokka. Toisin sanoen ekvivalenssiluokille määritetään todennäköisyysjakauma, joka vastaa alkioiden ositusta ekvivalenssiluokkiin. Kun koodauksen koossa tavoitellaan odotettua efektiivistä entropiaa, jokainen $y \in C$ esitetään käyttäen mahdollisimman lähelle $\log \frac{|A|}{|[y]_\sim|}$ bittiä.

Odotetulle efektiiviselle entropialle ja efektiiviselle entropialle pätee $H_o(A) \leq H_e(A)$. Yhtäsuuruus on voimassa, jos ja vain jos ekvivalenssiluokat ovat yhtäsuuret. Entropiat ovat siis vastaavassa suhteessa toisiinsa kuin Shannon-entropia ja pahimman tapauksen entropia.

Koodausta ja koodaavaa tietorakennetta kutsutaan minimaalisiksi, jos ne sisältävät vain tarvittavan informaation joukon K kyselyihin vastaamiseksi mille tahansa alkiolle $x \in A$. Jos $|C|$ on paljon pienempi kuin $|A|$, tilansäästö voi olla merkittävä verrattuna joukon A alkiot säilyttävään tietorakenteeseen. Minimaalisuuden saavuttaminen ei välttämättä ole helppoa ja saattaa vaatia vaikeiden kombinatoristen ongelmien ratkaisemista. Joskus voikin olla tarpeen osittaa A useampaan luokkaan kuin ekvivalenssirelaatiossa, jolloin joukosta C tulee minimaalista koodausta suurempi.

Koodausta ja koodaavaa tietorakennetta voi verrata häviölliseen tiivistykseen sikäli, että molemmissa alkuperäinen informaatio yleensä vähenee peruuttamattomasti. Informaatiota säilytetään sen verran, kuin käyttötarkoitus edellyttää.

Taulukko 3: Joukon $\{1, 2, 3, 4\}$ permutaatioiden ekvivalenssiluokat osavälin pienimmän alkion indeksin palauttaville kyselyille.

Luokkien koko	Luokkien määrä	Luokat
1	8	$\{1234\}$ $\{1243\}$ $\{1342\}$ $\{1432\}$ $\{2341\}$ $\{2431\}$ $\{3421\}$ $\{4321\}$
2	2	$\{1324, 1423\}$ $\{3241, 4231\}$
3	4	$\{2134, 3124, 4123\}$ $\{2143, 3142, 4132\}$ $\{2314, 2413, 3412\}$ $\{3214, 4213, 4312\}$

Tarkastellaan informaation vähenemistä permutaatioiden koodauksessa. Permutaatio [53] on bijektiivinen kuvaus joukolta itselleen. Kun joukossa on n alkiota, joukon permutaatioita on $n!$ kappaletta. Olkoon A joukon $\{1, 2, 3, 4\}$ permutaatioiden joukko [40]. Nyt $|A| = 4! = 24$. Merkitään yksittäistä permutaatiota lyhyesti lukujonolla. Esimerkiksi 1423 tarkoittaa alkioiden kuvautumista $1 \mapsto 1$, $2 \mapsto 4$, $3 \mapsto 2$ ja $4 \mapsto 3$, ja lukujonon alkioilla on indeksit yhdestä neljään. Olkoon K joukko osavälin minimi -kyselyjä, jotka palauttavat osavälin pienimmän alkion indeksin permutaatiota kuvaavassa lukujonossa. Siis $K = \{\texttt{rmq}(i, j) \mid 1 \leq i \leq j \leq 4\}$. Joukon koko $|K| = 10$. Permutaatiolla 1423 vaikkapa kysely $\texttt{rmq}(2, 4)$ palauttaa arvon 3, joka on alkion 2 indeksi.

Koska joukon K kyselyt palauttavat alkion arvon sijasta alkion indeksin, jokainen kysely palauttaa kahdella permutaatiolla saman vastauksen, jos ja vain jos jokaisen osavälin pienimmän alkion indeksi on permutaatioissa sama. Permutaatiot eli joukko A voidaan osittaa ekvivalenssiluokkiin, mikä on tehty taulukossa 3. Yhden alkion sisältäviä luokkia on kahdeksan, kahden alkion luokkia kaksi ja kolmen alkion luokkia neljä. Jokainen samaan luokkaan kuuluva permutaatio antaa samat vastaukset kaikkiin joukon K kyselyihin. Koodausten joukkoon C valitaan yksi permutaatio jokaisesta 14 luokasta.

Joukon A pahimman tapauksen entropia $H_p(A) = \log |A| = \log 24 \approx 4{,}58$ bittiä. Joukon A efektiiviseksi entropiaksi joukon K suhteen saadaan $H_e(A) = \log |C| = \log 14 \approx 3{,}81$ bittiä. Ekvivalenssiluokkien koot huomioon ottava joukon A odotettu efektiivinen entropia on $H_o(A) = -(8(\frac{1}{24} \log \frac{1}{24}) + 2(\frac{2}{24} \log \frac{2}{24}) + 4(\frac{3}{24} \log \frac{3}{24})) \approx 3{,}63$ bittiä. Joukon K kyselyihin vastaamiseen ei siten tarvita kaikkea alkuperäistä informaatiota.

Joukon $\{1, 2, \ldots, n\}$ permutaatioiden joukon ekvivalenssiluokkia on osoitettu olevan yhtä paljon kuin järjestyspuita, joissa on $n + 1$ solmua [40]. Todistus esitetään luvussa 4.1 osavälikyselyn toteutusajatuksen yhteydessä. Tuloksen vuoksi edellisen esimerkin ekvivalenssiluokkien määrä voidaan laskea samoin kuin järjestyspuiden määrä luvussa 3.3. Koska $n = 4$, ekvivalenssiluokkien määräksi saadaan

$$\frac{(2(n+1)-2)!}{(n+1)!\,((n+1)-1)!} = \frac{(2 \cdot 5 - 2)!}{5!\,(5-1)!} = 14.$$

Voidaan myös hyödyntää kaavaa, joka antaa arvion järjestyspuiden pahimman tapauksen entropialle. Edelliseen esimerkkiin sovellettuna se antaa arvion entropialle $H_e(A) = 2(n+1) - \Theta(\log(n+1)) = 10 - \Theta(\log 5)$ bittiä.

4 Tilaa säästävien tietorakenteiden käyttö

Edellisessä luvussa käsiteltiin tiedon entropiaa ja sen merkitystä tietorakenteiden tilantarpeelle. Seuraavaksi tutustutaan esimerkkinä osavälikyselyn tilaa säästävään toteutukseen koodaavalla tietorakenteella. Esimerkki havainnollistaa yleisiä koodaavien tietorakenteiden toteutustapoja sekä taulukon ja puiden välisiä yhteyksiä. Osavälikyselyllä on myös yhteyksiä LCE-ongelmaan. Sitä käytetään apurakenteena ratkaistaessa LCE-ongelma loppuosataulukon tai -puun avulla, kuten luvussa 2.2 jo todettiin. Lopuksi käsitellään LCE-ongelman tilaa säästäviä ratkaisuja, jolloin nähdään kytkös LCE-ongelman ja osavälikyselyn vähimmäistilavaativuuksissa.

4.1 Osavälikyselyn toteutusajatus

Osavälikyselyssä [49] halutaan vastaus johonkin kysymykseen taulukon osavälin alkioista. Kyselystä on monia muunnelmia. Voidaan etsiä tiettyä alkiota, alkion esiintymiskertoja, suurinta tai pienintä alkiota, tiettyä määrää suurimpia tai pienimpiä alkioita, tietyn kynnysrajan ylittävien tai alittavien alkioiden määrää tai jotakin arvoa lähinnä arvoltaan olevaa alkiota. Saatetaan etsiä myös tilastotietoa osavälin alkioista, kuten moodia, mediaania tai keskiarvoa. Kyselyn argumenteiksi voivat riittää osavälin päätepisteet, tai argumentteja voi olla useampia. Kyselyn paluuarvo voi olla yksi tai useampi alkio tai alkion sijasta sen indeksi. Osavälikyselyn voi yleistää yksiulotteisesta taulukosta esimerkiksi kaksiulotteiseen taulukkoon.

Osavälikyselyn sovelluksia on runsaasti [17]. Luvussa 2.2 kysely toimi loppuosataulukon ja -puun apurakenteena, joten kysely liittyy sitä kautta LCE-ongelman sovelluksiin sekä muihin näitä rakenteita käyttäviin algoritmeihin. Sovelluksia löytyy merkkijonoalgoritmeista, verkoista ja bioinformatiikasta.

Systemaattinen tietorakenne on hyvä vaihtoehto osavälikyselyn toteutukseen, jos vastaukseksi tarvitaan taulukon alkio, alkioita muuten tarvitaan tehtävän suoritukseen tai taulukko on pieni. Tällöin tietorakenteen kokoa yleensä minimoidaan hyödyntämällä taulukon sisältämää informaatiota. Koodaavaa tietorakennetta voi puolestaan käyttää selvitettäessä halutun alkion indeksiä, jolloin kaiken taulukon informaation tallentaminen on tarpeetonta. Indeksi riittää esimerkiksi tietyissä hakualgoritmeissa, joissa tekstidokumenttien kokoelmasta halutaan vastaukseksi lista hahmon sisältävistä dokumenteista. Indeksi riittää myös joissakin merkkijonon osavälille etsinnän rajoittavissa merkkijonohaun muunnelmissa.

Koodaavan tietorakenteen avulla on toteutettu viime vuosina useita osavälikyselyn muunnelmia [40]. Taulukon $A[1..n]$ kyselyihin on kehitetty muun muassa seuraavia ratkaisuja:

- Osavälin $A[i..j]$ maksimi- ja minimiarvojen indeksit löydetään $3n + o(n)$ bittiä käyttäen vakioajassa.
- Osaväliltä $A[i..j]$ löydetään k suurimman arvon indeksit $O(n \log k)$ bittiä käyttäen ajassa $O(k)$.
- Positiivisia ja negatiivisia lukuja sisältävältä osaväliltä $A[i..j]$ löydetään summan $\sum_{k=i'}^{j'} A[k]$ maksimoiva osaväli $A[i'..j']$ vakioajassa $O(n)$ bittiä käyttäen.
- Osaväliltä $A[i..j]$ löydetään yksi indeksi jokaiselle alkiolle, joka esiintyy osavälillä enemmän kuin $t(j - i + 1)$ kertaa, missä $0 < t < 1$. Tämä onnistuu $O(n \log(1/t))$ bittiä käyttäen ajassa $O(1/t)$ [20].
- Vain toisistaan eroavia arvoja sisältävästä taulukosta A löydetään indeksiä i lähin indeksi j, jolle $A[j] > A[i]$, vakioajassa $1{,}8n + o(n)$ bittiä käyttäen [24].

Katsotaan tarkemmin taulukon $A[1..n]$ osavälin pienimmän alkion indeksin palauttavaa kyselyä $\texttt{rmq}_A(i, j)$. Triviaali vakioaikainen ratkaisu on tallentaa kaikki vastaukset etukäteen tilassa $O(n^2)$ konesanaa. Ilman esikäsittelyä vastaaminen puolestaan onnistuu ajassa $O(n)$ käymällä osaväli läpi joka kyselyllä. Nämä ratkaisut eivät ole optimaalisia, sillä jo yli 30 vuotta on tunnettu vakioaikainen ratkaisu tilavaativuudeltaan $O(n)$ konesanaa [18].

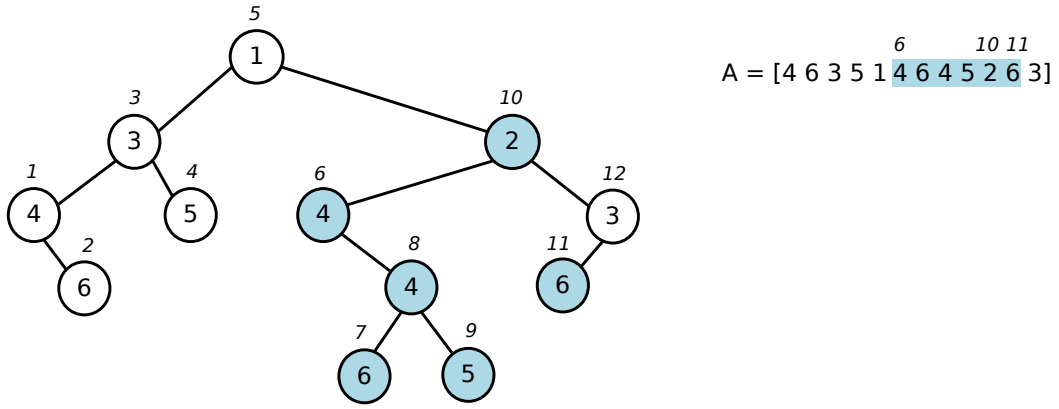
Tarkastellaan kyselyn $\texttt{rmq}_A$ toteuttamista koodaavalla tietorakenteella [15]. Kun alkioilla on järjestys, taulukosta A voi luoda karteesisen puun [53] käyttäen puun rekursiivista määritelmää:

1. Karteesisen puun juuri vastaa taulukon A pienintä alkiota. Olkoon pienimmän alkion indeksi k.
2. Juuren vasen lapsi on osavälin $A[1..k-1]$ karteesinen puu ja oikea lapsi osavälin $A[k + 1..n]$ karteesinen puu.

Jos osavälillä on enemmän kuin yksi pienin alkio, käytetään puuta luotaessa alkiota, jonka indeksi on pienin. Tällöin taulukkoa vastaava karteesinen puu on yksikäsitteinen.

Karteesisen puun solmujen ja taulukon A alkioiden välillä on bijektiivinen vastaavuus. Taulukko saadaan selville käymällä karteesinen puu läpi sisäjärjestyksessä: alkiota $A[k]$ vastaa puun solmu, jonka numero sisäjärjestyksessä on k. Lisäksi karteesisella puulla on minimikeko-ominaisuus: puun juuri vastaa taulukon A pienintä alkiota, ja jokaiselle alipuulle pätee, että sen juurta vastaavan alkion arvo on pienempi tai yhtä suuri kuin juuren lapsia vastaavien alkioiden arvot.

Karteesisen puun määritelmän ja keko-ominaisuuden seurauksena kysely $\texttt{rmq}_A(i, j)$ palautuu [18] alimman yhteisen esi-isän etsinnäksi taulukkoa A vastaavassa karteesisessa puussa. Olkoot u_i ja v_j puun solmut, jotka vastaavat osavälin $A[i..j]$ päätepisteiden alkioita, ja w näiden solmujen alin yhteinen esi-isä. Nyt solmun w numero



Kuva 4: Taulukko $A[1..12]$ ja sitä vastaava karteesinen puu [15]. Puun solmut on numeroitu sisäjärjestyksessä. Osavälikyselyyn $\texttt{rmq}_A(6, 11) = 10$ liittyvät alkiot ja niitä vastaavat solmut näkyvät sinisinä. Kyselyn palauttama arvo 10 on alkion 2 indeksi. Osavälikysely palautuu puussa alkioita $A[6]$ ja $A[11]$ vastaavien solmujen alimman yhteisen esi-isän etsinnäksi. Vastaus 10 on esi-isän numero sisäjärjestyksessä.

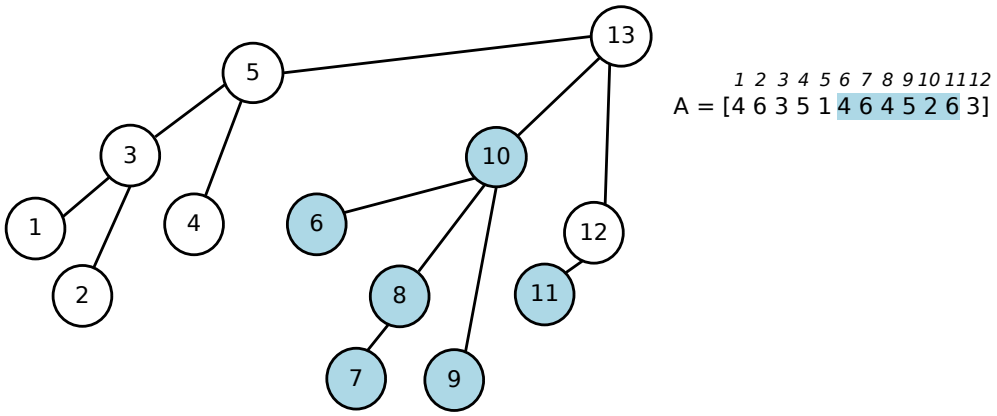
sisäjärjestyksessä on vastaus osavälikyselyyn. Kuvassa 4 on esimerkki taulukosta, sitä vastaavasta karteesisesta puusta ja osavälikyselystä.

Karteesinen puu on binääripuu. Sen voi muuntaa yksikäsitteisesti yleiseksi järjestyspuuksi [38], jossa solmun lapsien määrä ei ole rajoitettu. Karteesista puuta, jossa on n solmua, vastaa yleinen järjestyspuu, jossa on $n+1$ solmua. Se luodaan karteesisesta puusta esimerkiksi käyttäen rekursiivista määritelmää [15]:

1. Yleisen järjestyspuun juuri on uusi solmu, joka ei vastaa mitään karteesisen puun solmua.
2. Juuren lapsiksi vasemmalta oikealle tulevat solmut, jotka vastaavat karteesisen puun juuresta alkavaa oikeanpuoleisinta polkua. Tämän polun jokaisen solmun x vasemmasta alipuusta luodaan rekursiivisesti yleinen järjestyspuu, jonka juurena on solmu x.

Yleisessä järjestyspuussa taulukon A alkiot ovat jälkijärjestyksessä. Vaihtoehtoinen tapa [17] luoda yleinen järjestyspuu rekursiolla karteesisesta puusta on, että puun juuren lapsiksi vasemmalta oikealle tulevat karteesisen puun vasemmanpuoleisimman polun solmut ja näiden solmujen oikeanpuoleisista alipuista luodaan rekursiivisesti järjestyspuut. Tällöin taulukon A alkiot tulevat järjestyspuun solmuiksi esijärjestykseen. Tämä tapa kuitenkin mutkistaisi osavälikyselyn toteutusta hieman.

Kyselyyn $\texttt{rmq}_A(i, j)$ voi vastata ilman taulukon A alkioita käyttämällä karteesisesta puusta luotua yleistä järjestyspuuta [15]. Puu on nimittäin vasemmalta oikealle minimipuu (*left-to-right minima tree*). Vastaamiseksi lähdetään solmusta, jonka numero jälkijärjestyksessä on i, ja kuljetaan juurta kohti, kunnes saavutaan ensimmäiseen solmuun, jonka numero jälkijärjestyksessä on suurempi kuin j. Vastaus kyselyyn on polulla tätä edeltävän solmun numero jälkijärjestyksessä. Kuvassa 5 on esimerkki taulukkoa vastaavasta vasemmalta oikealle minimipuusta ja osavälikyselystä.



Kuva 5: Taulukkoa A vastaava yleinen järjestyspuu eli vasemmalta oikealle minimipuu, joka on luotu kuvan 4 karteesisesta puusta. Solmut on numeroitu jälkijärjestyksessä, jolloin juurta lukuun ottamatta solmun numero vastaa taulukon alkion indeksiä. Osavälikyselyn $\texttt{rmq}_A(6, 11) = 10$ liittyvät taulukon alkiot ja niitä vastaavat solmut näkyvät sinisinä. Puussa liikkuen vastaamiseen ei tarvita taulukon alkioiden arvoja. Vastaamiseksi lähdetään solmusta 6 ja kuljetaan kohti juurta, kunnes saavutaan ensimmäiseen solmuun, jonka numero on suurempi kuin 11. Kuljetaan siis polku $6 \rightsquigarrow 10 \rightsquigarrow 13$. Vastaus kyselyyn on solmua 13 polulla edeltävän solmun numero eli 10.

Kyselyyn $\texttt{rmq}_A(i, j)$ voi vastata vakioajassa $2n + o(n)$ bittiä käyttäen toteuttamalla taulukkoa A vastaavan vasemmalta oikealle minimipuun ja sen tarvittavat operaatiot. Kyseessä on koodaava tietorakenne, koska A vie tilaa $O(n \log \sigma)$ bittiä, missä σ on aakkoston koko, eikä vastaamiseen tarvita taulukon A alkioita. Tietorakenteen toteutusta käydään läpi seuraavassa luvussa.

Tarkastellaan osavälikyselyn entropiaa. Oletetaan, että taulukko $A[1..n]$ on kokonaislukujen permutaatio, P taulukon A permutaatioiden joukko ja K kyselyjen joukko $\{\texttt{rmq}_A(i, j) \mid 1 \leq i \leq j \leq n\}$. Joukolle P on olemassa ositus ekvivalenssiluokkiin, jossa joukon K kyselyt palauttavat samat vastaukset kaikille samaan luokkaan kuuluville permutaatioille. Nyt on olemassa bijektio ekvivalenssiluokkien joukolta taulukon A permutaatioiden vasemmalta oikealle minimipuiden joukolle [40]. Todistetaan tämä näyttämällä kolme seikkaa:

- Jokaisella permutaatiolla $p[1..n] \in P$ on vasemmalta oikealle minimipuu. Edellä kerrotusti taulukosta A ja siten mistä tahansa sen permutaatiosta voi luoda karteesisen puun avulla yleisen järjestyspuun, joka on haluttu puu. Siis väite pätee.

- Jokainen yleinen järjestyspuu, jossa on $n + 1$ solmua, on jonkin permutaation $p \in P$ vasemmalta oikealle minimipuu. Havaitaan järjestyspuussa jokaisen solmun paitsi juuren vastaavan pienintä alkiota permutaation osavälillä, jonka alkioita kyseisen solmun alipuun solmut vastaavat. Pienintä permutaation p alkiota vastaa järjestyspuun juuren vasemmanpuoleisin lapsi. Olkoon k lapsen numero jälkijärjestyksessä. Asetetaan $p[k] = 1$. Nyt permutaation voi

muodostaa rekursiivisesti asettamalla kokonaisluvut $[2..k-1]$ vastaamaan tämän lapsen alipuun solmuja ja kokonaisluvut $[k+1..n]$ järjestyspuun juuren muita lapsia ja niiden alipuiden solmuja. Solmujen jälkijärjestysarvojen avulla permutaation voi muodostaa ilman rekursiivista algoritmiakin kulkemalla puu läpi esijärjestyksessä. Juurta lukuun ottamatta solmuun saavuttaessa asetetaan $p[s] = i$, missä s on solmun numero jälkijärjestyksessä ja i saa kokonaislukuarvot väliltä $[1..n]$. Esimerkiksi kuvan 5 puu vastaa näin permutaatiota $p[1..12] = (3\ 4\ 2\ 5\ 1\ 7\ 9\ 8\ 10\ 6\ 12\ 11)$.

- Kahdella permutaatiolla on sama vasemmalta oikealle minimipuu, jos ja vain jos vastaukset kaikkiin joukon K kyselyihin ovat permutaatioilla samat. Näytetään tämä olettamalla ensin, että puut ovat samat. Koska kyselyihin vastaamiseen käytetään vain vasemmalta oikealle minimipuuta, vastauksetkin ovat tällöin samat. Oletetaan sitten vastausten olevan samat kahdella permutaatiolla. Tällöin kummankin permutaatioita vastaavan puun juuren vasemmanpuoleisin lapsi vastaa permutaation alkiota, jonka numero jälkijärjestyksessä on k, koska muuten permutaation väliä $[1..n]$ koskeva kysely palauttaisi eri vastauksen. Sama pätee rekursiivisesti vasemmanpuoleisimman lapsen alipuulle ja juuren muiden lasten alipuille, koska muuten jokin väliä $[1..k-1]$ tai väliä $[k+1..n]$ koskeva kysely palauttaisi permutaatioille eri vastauksen. Siis puut ovat samat.

Näin ollen osavälikyselyn ekvivalenssiluokat ja vasemmalta oikealle minimipuut vastaavat toisiaan yksi yhteen. Tämän seurauksena ekvivalenssiluokkien määrä saadaan Catalanin luvuista ja luokkien pahimman tapauksen entropia on $2n - \Theta(\log n)$, kuten luvussa 3.3 osoitettiin järjestyspuille. Siten $2n + o(n)$ bittiä on kooltaan lähes optimaalinen tietorakenne kyselyn $\texttt{rmq}_A$ toteutukseen.

Kyselyn $\texttt{rmq}_A$ toteutuksessa nähdään luvun 3.4 mukaiset koodaavan tietorakenteen luomisen ja käytön vaiheet. Taulukon A permutaatioita ei suoranaisesti jaeta ekvivalenssiluokkiin, mutta karteesinen puu ja yleinen järjestyspuu vastaavat rakenteiltaan jonkin ekvivalenssiluokan permutaatioita. Taulukon A voi tulkita luokkansa edustajaksi. Kun puun solmuihin ei tallenneta taulukon alkioiden arvoja, informaatio vähenee. Se kuitenkin riittää kyselyihin vastaamiseen, joten A jää tarpeettomaksi.

4.2 Osavälikyselyn toteutustapa

Edellisen luvun perusteella taulukon A kyselyyn $\texttt{rmq}_A$ voi vastata vasemmalta oikealle minimipuun eli yleisen järjestyspuun avulla ilman taulukon alkioita. Tarkastellaan puun ja tarvittavien operaatioiden toteutusta $2n + o(n)$ bittiä käyttäen.

Bittivektoria $B[1..n]$ käytetään apurakenteena yleisen järjestyspuun toteutuksessa, kuten useissa muissakin tilaa säästävissä tietorakenteissa. Bittivektorille on hyödyllistä määritellä kaksi operaatiota [29] jokaiselle $1 \leq i \leq n$:

$\texttt{rank}_{B,1}(i)$ palauttaa 1-bittien määrän välillä $B[1..i]$,

$\texttt{select}_{B,1}(j)$ palauttaa j:nnen 1-bitin indeksin, kun $1 \leq j \leq \texttt{rank}_{B,1}(n)$.

Taulukko 4: Esimerkki `rank`- ja `select`-operaatioista bittivektorille $B[1..8] =$ `01101011`.

i	0	1	2	3	4	5	6	7	8
$B[i]$		0	1	1	0	1	0	1	1
$\texttt{rank}_{B,1}(i)$	0	0	1	2	2	3	3	4	5
$\texttt{rank}_{B,0}(i)$	0	1	1	1	2	2	3	3	3
$\texttt{select}_{B,1}(i)$	0	2	3	5	7	8	9	9	9
$\texttt{select}_{B,0}(i)$	0	1	4	6	9	9	9	9	9

Määritellään myös $\texttt{rank}_{B,1}(0) = 0$, $\texttt{select}_{B,1}(0) = 0$ ja, jos $j > \texttt{rank}_{B,1}(n)$, niin $\texttt{select}_{B,1}(j) = n + 1$.

Määritellään `0`-biteille $\texttt{rank}_{B,0}(i)$ ja $\texttt{select}_{B,0}(j)$ vastaavasti kuin `1`-biteille. Taulukossa 4 on esimerkki operaatioista.

Jos bitti `1` tulkitaan sulkeeksi (ja bitti `0` sulkeeksi), bittivektorin B voi tulkita sulkeiden jonoksi. Määritellään operaatio $\texttt{excess}_B(i)$ palauttamaan avaavien sulkeiden määrä vähennettynä sulkevien sulkeiden määrällä välillä $B[1..i]$. Auki olevien sulkeiden määrä kohdassa $B[i]$ saadaan operaatiolla

$$\texttt{excess}_B(i) = \texttt{rank}_{B,1}(i) - \texttt{rank}_{B,0}(i).$$

Määritellään lisäksi $\texttt{excess}_B(0) = 0$.

B kuvaa tasapainoisia sulkeita [29], jos $\texttt{excess}_B(i) \geq 0$ kaikilla $1 \leq i \leq n$ ja $\texttt{excess}_B(n) = 0$. Tällöin n on parillinen ja `1`- ja `0`-bittejä on yhtä monta. Jokaista avaavaa sulkua kohdassa $B[i]$ vastaa sulkeva sulku kohdassa $B[j]$, missä $j > i$. Lisäksi avaavaa sulkua välillä $B[i + 1..j - 1]$ vastaa sulkeva sulku samalla välillä. Kutsutaan näiden sulkeiden muodostamaa väliä segmentiksi. Kaksi segmenttiä joko ovat täysin erilliset tai toinen niistä sisältyy toiseen, joten segmentit muodostavat hierarkian sisältyvyysrelaation suhteen.

Kyselyn $\texttt{rmq}_A$ toteuttamiseksi taulukkoa $A[1..n]$ vastaava yleinen järjestyspuu esitetään tasapainoisina sulkeina. Bittivektori $B[1..2n + 2]$ saadaan käymällä puu läpi syvyysjärjestyksessä. Bittivektoriin lisätään `1` saavuttaessa solmuun ensimmäisen kerran ja `0` poistuttaessa solmusta viimeisen kerran. Bittivektorissa jokainen segmentti kuvaa alipuuta. Välivaiheena ei ole välttämätöntä luoda puurakennetta, sillä taulukosta voi pinon avulla luoda bittivektorin myös suoraan [40] ajassa $O(n)$.

Tasapainoisilla sulkeilla esitettyä puuta voi hyödyntää monipuolisesti toteuttamalla bittivektorille B yleisluontoisia operaatioita [11]. Seuraavat kaksi operaatiota palauttavat kohdasta i eteenpäin tai taaksepäin lähimmän indeksin, jossa puun syvyys kohdan i suhteen vastaa argumenttia d:

$$\texttt{fwdsearch}_B(i, d) = \min\{j > i \mid \texttt{excess}_B(j) = \texttt{excess}_B(i) + d\},$$

$$\texttt{bwdsearch}_B(i, d) = \max\{j < i \mid \texttt{excess}_B(j) = \texttt{excess}_B(i) + d\}.$$

Jos suhteellista syvyyttä d ei löydy haetulta väliltä, määritellään erikoistapauksina $\texttt{fwdsearch}_B(i, d) = n + 1$ ja $\texttt{bwdsearch}_B(i, d) = 0$.

Nyt saadaan selville esimerkiksi kohdan i avaavaa sulkua vastaava sulkeva sulku operaatiolla $\texttt{fwdsearch}_B(i, -1)$. Kohdan j sulkevaa sulkua vastaava avaava sulku selviää operaatiolla $\texttt{bwdsearch}_B(j, 0) + 1$. Operaatio $\texttt{bwdsearch}_B(j, -2) + 1$ puolestaan palauttaa kohdan j avaavalle sulkeelle segmenttien hierarkiassa ylemmän segmentin avaavan sulun eli solmun vanhemman.

Algoritmissa 1 esitetään kyselyn $\texttt{rmq}_A(i, j)$ toteutus [40]. Algoritmissa käytetään bittivektoria B, joka kuvaa taulukon A vasemmalta oikealle minimipuun tasapainoisina sulkeina. Riveillä 1 ja 2 etsitään bittivektorista kyselyn osaväli, eli tallennetaan apumuuttujiin osavälin alkua ja loppua kuvaavien solmujen sulkevien sulkeiden indeksit. Osaväliltä halutaan löytää puussa matalimmalla syvyydellä oleva solmu. Siis etsitään osaväliltä kohta, jossa on vähiten auki olevia sulkeita. Teknisenä toteutuksena etsitään rivillä 3 ensin $\texttt{minexcess}_B$-operaatiolla osavälin pienin $\texttt{excess}_B$-arvo suhteessa osavälin alkua edeltävään indeksiin $p - 1$. Rivillä 4 haetaan arvoa vastaava osavälin indeksi r. Rivillä 5 etsitään `0`-bittien määrä välillä $B[1..r]$. Saatu vastaus on halutun solmun indeksi taulukossa A eli vastaus kyselyyn $\texttt{rmq}_A(i, j)$. Kuvassa 6 on esimerkki puusta ja kyselystä.

Algoritmi 1 Kyselyn $\texttt{rmq}_A(i, j)$ toteutus tasapainoisten sulkeiden B avulla

Syöte: taulukon A osavälin $[i..j]$ indeksit i ja j
Tuloste: osavälin pienimmän alkion indeksi

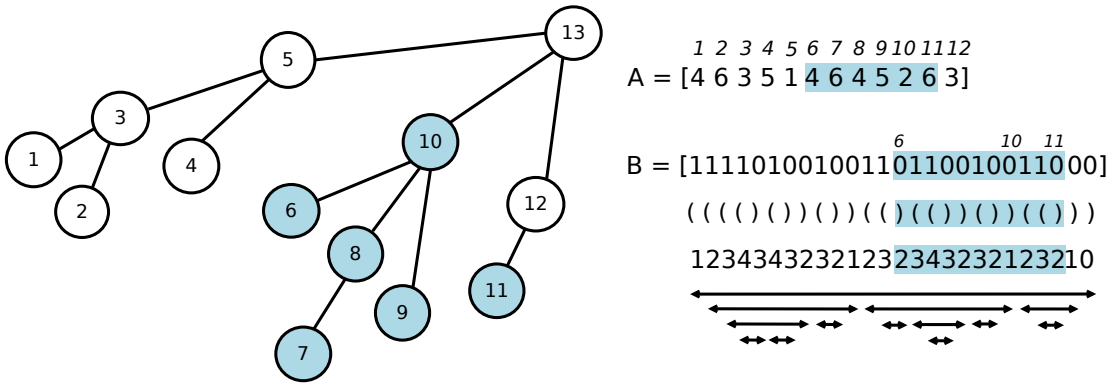
1: $p \leftarrow \texttt{select}_{B,0}(i)$
2: $q \leftarrow \texttt{select}_{B,0}(j)$
3: $m \leftarrow \texttt{minexcess}_B(p, q)$
4: $r \leftarrow \texttt{fwdsearch}_B(p - 1, m)$
5: **return** $\texttt{rank}_{B,0}(r)$

Edellä mainitut bittivektorin $B[1..n]$ operaatiot on mahdollista toteuttaa vakioajassa $o(n)$ lisäbittiä käyttäen. Tarkastellaan `rank`- ja `fwdsearch`-operaatioita esimerkkeinä tilaa säästävien tietorakenteiden toteutustekniikoista.

Operaation $\texttt{rank}_{B,1}(i)$ triviaali toteutus on käydä läpi $B[1..i]$ ja laskea `1`-bittien määrä joka suorituskerralla. Tapa vie vakiotilan mutta aikaa $O(n)$. Toisaalta voi luoda etukäteen kokonaislukutaulukon $R[1..n]$, jolle pätee $R[i] = \texttt{rank}_{B,1}(i)$. Nyt vastaus saadaan taulukosta vakioajassa, mutta taulukon tilavaativuus $\Omega(n \log n_1)$ bittiä, missä n_1 on `1`-bittien määrä bittivektorissa, on yleensä suurempi kuin bittivektorin. Kolmas toteutustapa käyttää hyväksi huomiota, että $\texttt{rank}_{B,1}(i)$ on välin $B[1..i]$ osasumma, kun `1`-bitit tulkitaan kokonaisluvuiksi 1. Osasumman avulla [40] $\texttt{rank}_{B,1}(i)$-operaation voi toteuttaa esimerkiksi n lisäbittiä käyttäen aikavaativuusluokassa $O(\log n)$.

Tilavaativuudeltaan $o(n)$ lisäbitin toteutusta [42] varten oletetaan selkeyden vuoksi, että n on arvon $\log n$ monikerta. Jaetaan B käsitteellisesti osiin ylilohkoiksi, joista kunkin pituus $l = \log^2 n$. Tallennetaan taulukkoon $R[1..\lfloor n/l \rfloor + 1]$ `1`-bittien määrä jokaisen ylilohkon alkua edeltävän indeksin kohdalla, toisin sanoen

$$R[i] = \texttt{rank}_{B,1}((i - 1)l).$$



Kuva 6: Taulukko $A[1..12]$ ja sitä vastaava vasemmalta oikealle minimipuu solmut jälkijärjestyksessä numeroituna. $B[1..26]$ esittää puun bittivektorina, joka näkyy myös tasapainoisiksi sulkeiksi tulkittuna. Segmenttien hierarkia ja vastaava lukujono kertovat, millä syvyydellä puussa ollaan. Kyselyyn $\texttt{rmq}_A(6, 11) = 10$ liittyvät alkiot ja solmut näkyvät sinisinä. Bittivektorissa osaväli alkaa taulukon osavälin vasemmanpuoleisinta alkiota vastaavan solmun sulkevasta sulusta ja päättyy oikeanpuoleisinta alkiota vastaavan solmun sulkevaan sulkuun. Kyselyn vastaus saadaan etsimällä bittivektorista osavälin solmu, joka on puussa alimmalla syvyydellä.

Jaetaan sitten käsitteellisesti jokainen ylilohko lohkoiksi, joiden pituus $l' = \log n$. Tallennetaan taulukkoon $R'[1..n/l' + 1]$ `1`-bittien määrä jokaisen lohkon alkua edeltävän indeksin kohdalla vastaavan ylilohkon alusta laskettuna. Toisin sanoen

$$R'[i] = \texttt{rank}_{B,1}((i-1)l') - R[\lfloor (i-1)l'/l \rfloor + 1].$$

Taulukossa R on $\lfloor n/l \rfloor + 1 = \lfloor n/\log^2 n \rfloor + 1$ alkiota, joista kukin vie tilaa $\log n$ bittiä. Näin ollen taulukon R tilavaativuus bitteinä on

$$\left(\left\lfloor \frac{n}{\log^2 n} \right\rfloor + 1 \right) \log n \leq \left(\frac{n}{\log^2 n} + 1 \right) \log n = \frac{n}{\log n} + \log n = o(n).$$

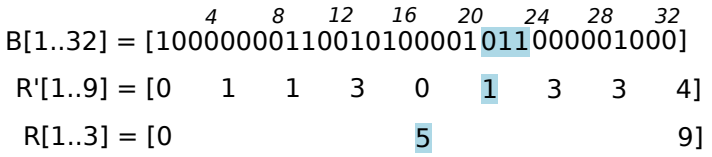
Taulukossa R' on $n/l' + 1 = n/\log n + 1$ alkiota, mutta alkion suurin mahdollinen arvo on $l - l'$, joka vie tilaa $1 + \log(l - l')$ bittiä. Taulukon R' tilavaativuudeksi bitteinä saadaan

$$\left(\frac{n}{\log n} + 1 \right) (1 + \log(l - l')) = \frac{n}{\log n} + \frac{n \log(l - l')}{\log n} + 1 + \log(l - l') = o(n).$$

Summassa toisen ja neljännen termin tilavaativuudeksi nähdään $o(n)$ bittiä havaitsemalla, että funktioille $f(n) = \log n$ ja $f(n) = n$ pätee

$$\lim_{n \to \infty} \frac{\log(l - l')}{f(n)} = \lim_{n \to \infty} \frac{\log(\log^2 n - \log n)}{f(n)} = 0.$$

Kuvassa 7 on esimerkki $\texttt{rank}_{B,1}(i)$-operaatiosta taulukoiden R ja R' avulla. Operaatio ei ole vakioaikainen, koska se vaatii kahden taulukkohaun lisäksi `1`-bittien



Kuva 7: Bittivektorin $B[1..32]$ täydentäminen $o(n)$ bitin taulukoilla `rank`-operaation toteuttamiseksi. Havainnollisuuden vuoksi tässä ylilohkon pituus $l = 16$ bittiä, lohkon pituus $l' = 4$ bittiä ja taulukoiden R ja R' alkiot on ryhmitelty allekkain bittivektorin alkioiden kanssa. Taulukkoon R on tallennettu bittivektorin alusta laskettu `1`-bittien määrä jokaisen ylilohkon alkua edeltävän indeksin kohdalla ja taulukkoon R' `1`-bittien määrä jokaisen lohkon alkua edeltävän indeksin kohdalla vastaavan ylilohkon alusta lukien. Sininen väri kuvaa operaatiota $\texttt{rank}_{B,1}(23) = 8$. Vastaus saadaan laskemalla yhteen $R[2]$, $R'[6]$ ja `1`-bittien määrä osavälillä $B[21..23]$.

laskemisen lohkon sisältä. Laskentaa kutsutaan `popcount`-operaatioksi, ja siihen on olemassa [34] ajassa $O(n_1)$ toimivia algoritmeja, missä n_1 on `1`-bittien määrä läpikäytävällä välillä. Lisäksi useat laitteistot tukevat operaation tehokasta toteutusta [40]. Mutta vaihtoehtona on tehdä `rank`-operaatio vakioaikaiseksi tallentamalla taulukkoon `1`-bittien määrä jokaisessa bittijonossa, jonka pituus on $(\log n)/2$ bittiä. Bittijonoja on $2^{(\log n)/2} = \sqrt{n}$ kappaletta, mistä saadaan taulukon tilavaativuudeksi $O(\sqrt{n} \log \log n) = o(n)$ bittiä. Taulukon avulla `1`-bittien määrä välillä, jonka pituus on $O(\log n)$ bittiä, saadaan vakioajassa. Käytännössä `popcount`-operaatio voi kuitenkin olla nopeampi.

Sovellusten nopeuttamiseksi lohkon ja ylilohkon pituudet kannattaa valita siten, että muistista voidaan lukea kerrallaan kokonainen tavu tai konesana. Lisäksi taulukot R ja R' voi tallentaa muistiin lomittain, jolloin haettava tieto on mahdollisimman usein välimuistissa. Taulukot $B[1..n]$, R ja R' voi toteuttaa tehokkaasti esimerkiksi $1{,}375n$ bittiä käyttäen, jos konesanan pituus on 32 tai 64 bittiä. Tilantarvetta voi vähentää käyttämällä taulukoiden R ja R' lisäksi kolmatta taulukkoa yliylilohkoja varten.

`rank`-operaation tilantarvetta voi vähentää [43] tiivistämällä bittivektori B. Tällöin B vie tilaa $nH_s(B) + o(n)$ bittiä, ja aputietorakenteet voi edelleen toteuttaa $o(n)$ lisäbitillä [40]. Tiivistetty bittivektori vie tilaa 64 bitin konesanalla noin $0{,}125n$ bittiä yli Shannon-entropian. `rank`-operaatio on käytännössä edelleen nopea, vaikka sen teoreettiseksi aikavaativuudeksi tulee $O(\log n)$. Tiivistämisessä voi käyttää myös kontekstin huomioon ottavia tapoja, jolloin B vie tiivistettynä tilaa $nH_k(B) + o(n)$ bittiä, missä $H_k(B)$ on kontekstissa laskettu entropia ja kontekstin pituus $k > 0$. On kehitetty myös tapoja käsitellä tehokkaasti harvoja bittivektoreita, joissa `1`-bittien määrä on paljon pienempi kuin `0`-bittien tai päinvastoin. Hybriditiivistyksessä [31] puolestaan bittivektori jaetaan lohkoihin, joista kullekin valitaan sopivin tiivistysmenetelmä.

Kaikkia kulloinkin tarvittavia operaatioita varten ei välttämättä tarvitse toteuttaa omia aputietorakenteita. Jos operaatio $\texttt{rank}_{B,1}(i)$ on toteutettu, $\texttt{rank}_{B,0}(i) = i - \texttt{rank}_{B,1}(i)$. Toisena esimerkkinä huomataan `rank`- ja `select`-operaatioiden yhteys

$\texttt{rank}_{B,1}(\texttt{select}_{B,1}(j)) = j$. Saadaan kaava $\texttt{select}_{B,1}(j) = \min\{i \mid \texttt{rank}_{B,1}(i) = j\}$. Tämän operaation aikavaativuus on $O(\log n)$, jos `rank` on vakioaikainen. Koska `rank`-operaation arvot muodostavat kasvavan jonon, `select`-operaation arvo löydetään jonosta binäärihaulla. `rank`- ja `select`-operaatiot voi toteuttaa tehokkaasti myös aputietorakenteet yhdistäen [55] käyttämällä hyväksi operaatioiden välistä yhteyttä.

Kokonaan erikseen bittivektorin B `select`-operaation voi toteuttaa [10] vakioajassa $o(n)$ lisäbittiä käyttäen. Toteutusajatus on sama kuin `rank`-operaatiolla, eli B jaetaan sopivasti pienempiin osiin, joihin liittyviä `select`-operaation arvoja taulukoidaan.

Siirrytään `fwdsearch`-operaatioon. Se toteutetaan luomalla järjestyspuuta tasapainoisina sulkeina kuvaavasta bittivektorista B osavälin min–max -puu eli rmM-puu (*range min-max tree*) [11]. Puu vie tilaa $o(n)$ lisäbittiä. Puun avulla voi toteuttaa laajan joukon muitakin operaatioita, joista monet vaativat ennen puun kehittämistä omia tietorakenteitaan. Operaatioilla voi muun muassa liikkua bittivektorin kuvaamassa järjestyspuussa tietyn solmun suhteen, löytää tietty esi- tai jälkijärjestyksen solmu, selvittää alipuun koko tai lehtien määrä ja löytää kahden solmun alin yhteinen esi-isä. Useimmat operaatiot toteutetaan palauttamalla ne `fwdsearch`- ja `bwdsearch`-operaatioiksi tai muutamaksi muuksi operaatioksi.

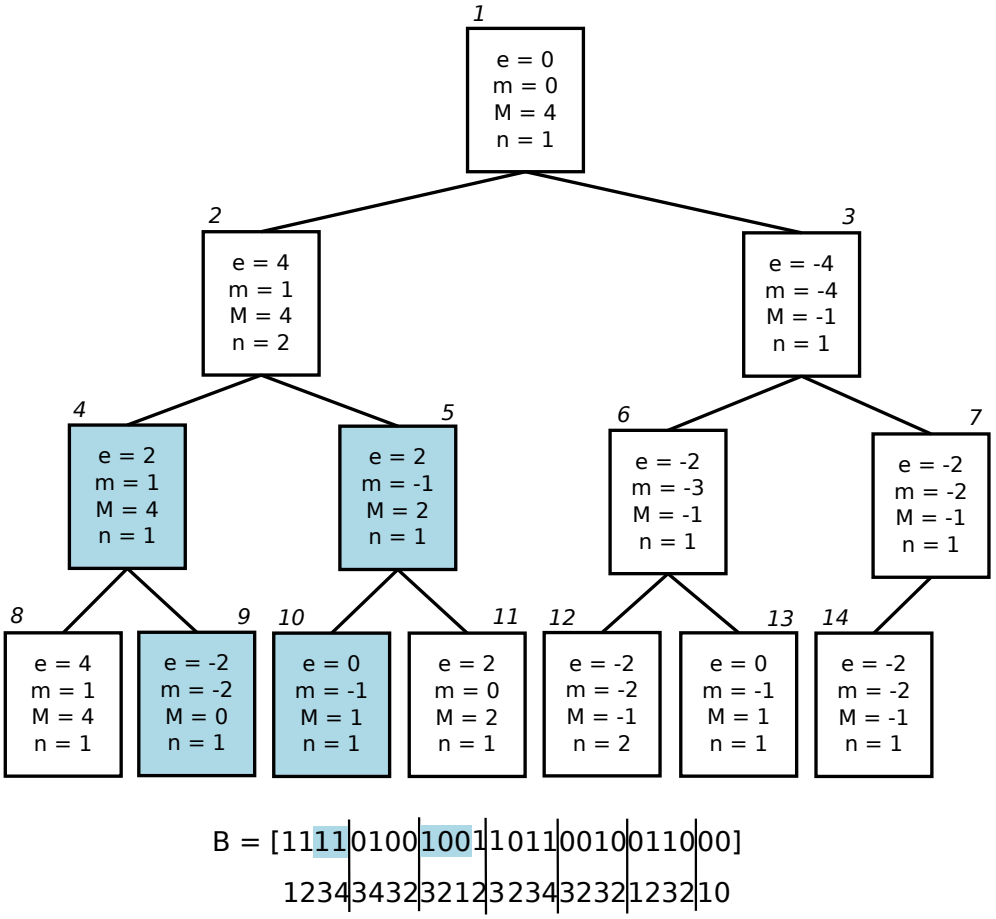
Tarkastellaan rmM-puun yksinkertaista toteutusta, jossa operaatioiden aikavaativuudeksi tulee $O(\log n)$ [40]. rmM-puu on täydellinen binääripuu, jonka kaikki tasot ovat täynnä. Mahdollisena poikkeuksena on alin taso, jossa solmut ryhmitellään vasemmalle. Bittivektori B jaetaan lohkoihin, joiden pituus on b bittiä. Puun vasemmalta lukien k. lehti kattaa välin $B[(k-1)b+1..kb]$, ja sisäsolmut puolestaan kattavat kummankin lapsensa välien yhdisteen. Solmuihin tallennetaan toteutettavien operaatioiden mukaan vaihtelevia tietoja. Usein on hyödyllistä tallentaa välin $B[i..j]$ kattavaan solmuun seuraavat tiedot:

$$e = \texttt{excess}_B(j) - \texttt{excess}_B(i-1),$$
$$m = \min\{\texttt{excess}_B(p) - \texttt{excess}_B(i-1) \mid i \le p \le j\},$$
$$M = \max\{\texttt{excess}_B(p) - \texttt{excess}_B(i-1) \mid i \le p \le j\},$$
$$n = |\{p \mid i \le p \le j, \texttt{excess}_B(p) - \texttt{excess}_B(i-1) = m\}|.$$

Täten e on syvyyden muutos välillä, m välin paikallinen syvyysminimi vasemmalta lukien, M välin paikallinen syvyysmaksimi vasemmalta lukien ja n paikallisen syvyysminimin esiintymiskertojen määrä välillä.

$\texttt{fwdsearch}_B$-operaatio koostuu kolmesta osasta ja käyttää solmujen tietoja e ja m. Ensin etsitään kohdasta $B[i+1]$ alkaen kyseisen lohkon loppuosa. Jos vastausta ei löytynyt, etsitään rmM-puun avulla vastauksen sisältävä lohko. Sen sisältä paikallistetaan vastaus.

Kuvassa 8 on esimerkki rmM-puusta ja operaatiosta $\texttt{fwdsearch}_B(2,-1) = 11$. Etsitään siis pienin $j > 2$, jolla $\texttt{excess}_B(j) = \texttt{excess}_B(2) - 1 = 1$. $B[3]$ kuuluu solmua 8 vastaavaan lohkoon. Ensin käydään läpi lohkon loppuosa eli osaväli $B[3..4]$. Vastaus-



Kuva 8: Osavälin min–max -puu lohkon koolla $b = 4$. Solmujen numerot vastaavat solmujen järjestystä taulukossa, johon puu tallennetaan. Bittivektori B esittää kuvan 6 järjestyspuuta ja lukujono sen alla järjestyspuun syvyyttä. Operaatioon $\texttt{fwdsearch}_B(2, -1) = 11$ liittyvät alkiot ja solmut näkyvät sinisinä.

ta ei löytynyt, ja lohkon lopussa suhteellinen syvyys $d' = 2$ kohtaan $B[2]$ nähden. Solmu 8 on parillinen ja siten vasen lapsi. Tarkistetaan, onko vastaus oikean lapsen eli solmun 9 lohkossa laskemalla $d' + m = 2 - 2 = 0$, missä m saadaan solmusta 9. Koska $0 > -1$, vastaus ei ole lohkossa. Päivitetään arvoksi $d' = 0$, ja siirrytään vanhempaan 4. Tarkistetaan, onko vastaus sisarsolmun 5 kattamalla välillä. Vastaus löytyy, koska nyt $d' + m = 0 - 1 = -1$. Siirrytään solmuun 5. Huomataan vastauksen olevan vasemman lapsen eli solmun 10 lohkossa, koska solmussa $m = -1$. Käydään läpi osaväliä $B[9..12]$ alusta lukien, ja löydetään kohta $B[11]$, jossa $d' = -1$.

rmM-puun avulla voi toteuttaa algoritmissa 1 esiintyneen $\texttt{minexcess}_B$-operaation [40]. Toteutus koostuu $\texttt{fwdsearch}_B$-operaation tavoin kolmesta osasta, joissa tutkitaan yksittäisen lohkon bittejä tai liikutaan puussa tarkastellen solmuihin tallennettuja arvoja. Operaation toteutusta ei selosteta tässä tarkemmin.

rmM-puun voi luoda käymällä ensin bittivektori B läpi lineaarisesti ja laskemalla lehtiin tallennettavat arvot. Sen jälkeen sisäsolmujen tiedot voi laskea rekursiivisesti

lehtien avulla. Puussa on $\lceil n/b \rceil$ lehteä. Täydellinen binääripuu voidaan tallentaa taulukkoon ilman osoittimia, joten koko rmM-puu mahtuu taulukkoon $R[1..2\lceil n/b \rceil - 1]$. Kohtaan $R[1]$ tallennetaan juuri, jota seuraavat muut solmut tasoittain vasemmalta oikealle. Kohdassa $R[i]$ sijaitsevan solmun vasen lapsi on kohdassa $R[2i]$ ja oikea lapsi kohdassa $R[2i+1]$. Solmun vanhempi on kohdassa $R[\lfloor i/2 \rfloor]$.

rmM-puun solmu vie tilaa $O(\log n)$ bittiä olettaen, että solmuun tallennetaan vakiomäärä tietoja, joista kukin vie tilaa $\log n$ bittiä. Näin koko puu vie tilaa $O((n/b) \log n)$ bittiä.

Operaatioiden nopeuttamiseksi luodaan rmM-puun lisäksi taulukko $C[1..2^c]$. Valitaan $c = \lfloor (\log n)/2 \rfloor$, jolloin taulukossa on enintään $2^{(\log n)/2} = \sqrt{n}$ alkiota. Taulukkoon tallennetaan jokaisesta bittijonosta, jonka pituus on c, tiedot e, m, M ja n kuten rmM-puussa. Siis alkiossa $C[i]$ on tiedot kokonaislukua $i-1$ vastaavasta bittijonosta. Taulukon tilavaativuus on $O(\sqrt{n} \log n)$ bittiä ja luomisen aikavaativuus $O(\sqrt{n} \log n)$.

Taulukosta C saadaan vakioajassa tiedot lohkosta, jonka pituus on c bittiä. Taulukon avulla voi nopeuttaa myös rmM-puun luomista, jos lohkon b pituudeksi valitaan arvon c monikerta. Tällöin tietojen laskenta vie aikaa lehtisolmuissa $O(n/\log n)$ ja sisäsolmuissa $O(n/b)$.

Edellä esitetysti bittivektorin $B[1..n]$ operaatioiden toteutus vaatii bittivektorin lisäksi rmM-puun taulukkona R ja taulukon C. Jos valitaan arvoiksi $b = \Theta(\log^2 n)$ ja $c = \Theta(\log n)$, saadaan tilavaativuudeksi yhteensä $n + O(n/\log n) = n + o(n)$ bittiä. rmM-puun ja taulukon C luominen vie aikaa yhteensä $O(n/\log n)$. Tämän jälkeen operaatiot voi suorittaa ajassa $O(\log n)$.

Käytännössä arvoksi c kannattaa valita esimerkiksi 8 tai 16, jonka kokoisten bittijonojen lukeminen muistista on nopeaa. Lohkon b koko kannattaa valita välimuistin käyttö optimoiden, esimerkiksi 1024 bittiä on sopiva moniin sovelluksiin.

rmM-puusta on olemassa monimutkaisempi toteutus [11], jonka avulla operaatioiden aikavaativuus on $O(\log \log n)$ ja tilavaativuus edelleen $O(n/\log n)$ bittiä. Bittivektorin B jokaisesta osasta, jonka pituus on $\Theta(\log^3 n)$ bittiä, luodaan oma rmM-puu lohkon koolla $b = \log n \log \log n$. Operaatiot suoritetaan puun sisällä samalla tavalla kuin yhden puun mallissa. Vähintään kahta puuta käyttävät operaatiot ovat vaikeampia. Niitä varten tallennetaan puukohtaisia lisätietoja siten, ettei tilavaativuus kuitenkaan muutu.

Usean rmM-puun malli on tällä hetkellä nopein vaihtoehto sovelluksiin. rmM-puun voi toteuttaa myös vakioaikaisia operaatioita tukevalla tavalla $o(n)$ bittiä käyttäen [41], mutta operaatioiden suoritusaika sisältää tällöin suuren vakiokertoimen. Lisäksi puu käyttää apurakenteita, joita on vaikea toteuttaa tehokkaasti.

4.3 Ongelman ratkaisuja

Edellisessä luvussa tutustuttiin tilaa säästävien tietorakenteiden toteutukseen. Taulukot ja puut ovat keskeisiä tietorakenteita. Bittivektorin `rank`- ja `select`-operaatiot

ja osavälin min–max -puu ovat esimerkkejä yleiskäyttöisistä välineistä. Niiden toteutuksissa toistuva piirre on sopivan otoksen poiminta ja tallentaminen alkuperäisestä syötteestä, jotta halutut aika- ja tilavaativuudet saavutetaan. Välineiden lisäksi ongelmien ratkaiseminen edellyttää sopivaa toteutusajatusta. Vakioaikaisen, tilavaativuudeltaan lähes optimaalisen ratkaisun saavuttaminen ei välttämättä ole yksinkertaista, kuten havaittiin osavälikyselyn toteutuksessa. Toisaalta aikavaativuudeltaan nopein ratkaisu voi olla käytännössä hitaampi verrattuna yksinkertaisempaan ratkaisuun samassa tilavaativuusluokassa.

Ratkaistaessa LCE-ongelma ilman tilaa säästäviä tietorakenteita nähtiin luvussa 2.2 aika- ja tilavaativuuksien vaihtokauppa: ongelma ratkeaa vakioajassa tilassa $O(n)$ konesanaa tai vakiotilassa ajassa $O(n)$. Tarkastellaan aika- ja tilavaativuuden yhteyttä tarkemmin, sillä LCE-ongelmaan on kehitetty 2010-luvulla useita tilaa säästäviä ratkaisuja.

Aikavaativuutta voi mitata konesanan kokoisten muistialkioiden luku- ja kirjoituskertojen määrällä (*non-uniform cell probe model*) [8]. Muuta laskentaa ei oteta huomioon, ja algoritmi voi vaihdella erikokoisilla syötteillä. Muistialkion oletetaan sisältävän yhden syötteen alkion, esimerkiksi merkkijonon merkin. Mallissa laitteiston käskykannalla ei ole merkitystä, kunhan jokainen käsky operoi korkeintaan vakiomäärällä muistialkioita. Mallissa saadut tulokset rinnastuvat tutkielmassa oletettuun hajasaantimalliin, jos jokaista muistialkion luku- tai kirjoituskertaa kohti suoritetaan enintään vakiomäärä muuta laskentaa.

Mallissa on osoitettu mille tahansa n alkion syötteelle pätevä `lce`-kyselyn aika–tilavaihtokauppa: systemaattisella tietorakenteella, jonka koko on $O(n/t)$ bittiä, toteutettu kysely vie aikaa $\Omega(t)$, missä $1 \leq t \leq n$. Vastaava tulos todistettiin ensin yleisesti taulukon A osavälikyselylle $\texttt{rmq}_A(l, r)$. Tulos pätee näin erityisesti bittivektorille B, ja tulos osoitettiinkin `lce`-kyselylle palauttamalla $\texttt{rmq}_B$-kysely $\texttt{lce}_B$-kyselyksi. Palautus esitetään seuraavaksi, sillä se havainnollistaa `lce`- ja `rmq`-kyselyjen välistä yhteyttä.

Millä tahansa $\texttt{lce}_B$-kyselyn toteuttavalla tietorakenteella voi toteuttaa $\texttt{rmq}_B(l, r)$-kyselyn tekemällä yhden $\texttt{lce}_B$-kyselyn ja käyttämällä vakiomäärän lisätilaa. Tallennetaan ensin bittivektorin B pisimmän pelkästään `1`-bittejä sisältävän välin $B[i..j]$ indeksit i ja j. Välin pituus $z = j - i + 1$. Olkoon $p = \texttt{lce}_B(l, i)$. Nyt

$$\texttt{rmq}_B(l, r) = \begin{cases} l + p & \text{jos } p \leq z \text{ ja } l + p \leq r \\ l + z & \text{jos } p > z \text{ ja } l + z \leq r \\ l & \text{muuten.} \end{cases}$$

Ensimmäisen kohdan todistamiseksi oletetaan $p \leq z$. Koska $B[i..j]$ sisältää vain `1`-bittejä, niin $B[l..l + p - 1]$ sisältää vain `1`-bittejä, ja $B[l + p] = \texttt{0}$. Nyt jos $l + p \leq r$, niin $B[l + p]$ on välin $B[l..r]$ minimi. Jos puolestaan $l + p > r$, niin $B[l..r]$ sisältää vain `1`-bittejä. Tällöin palautetaan viimeisen kohdan mukaisesti l (paluuarvoksi käy oikeastaan mikä tahansa välin $B[l..r]$ indeksi).

Toisen kohdan todistamiseksi oletetaan $p > z$. Nyt $B[l..l + z - 1]$ sisältää vain `1`-

bittejä ja $B[l+z] = \texttt{0}$. Jos $l+z \leq r$, niin $B[l+z]$ on välin $B[l..r]$ minimi. Jos puolestaan $l+z > r$, niin $B[l..r]$ sisältää vain `1`-bittejä. Tällöin palautetaan taas l.

Koska kyselyn $\texttt{rmq}_B$ voi näin palauttaa $\texttt{lce}_B$-kyselyksi, aika–tila-vaihtokauppaa koskeva tulos pätee myös `lce`-kyselyyn. Koska mallissa oletetaan muistialkion sisältävän yhden syötteen alkion, tulos pätee aakkoston koolla $|\sigma| \leq 2^w$, missä w on konesanan koko bitteinä. Tällöin yksittäinen merkkijonon merkki mahtuu konesanaan.

Olkoot merkkijono $S[1..n]$, $\texttt{lce}_S$-kyselyn toteuttavan systemaattisen tietorakenteen koko $S(n)$ bittiä ja kyselyn aikavaativuus $T(n)$. Edellinen tulos osoittaa tietorakenteella, jonka koko on $O(n/t)$ bittiä, $\texttt{lce}_S$-kyselyn aikavaativuudeksi $\Omega(t)$. Siis $S(n)T(n) = \Omega(n)$. Oletetaan mallin edellä mainittujen rajoitusten lisäksi, että merkkijono S on vain luettavissa, aakkoston koko on vähintään $2^{8\lceil S(n)/n \rceil}$ ja $S(n) = \Omega(n)$ bittiä. Tällöin on osoitettu [36] tulos $S(n)T(n) = \Omega(n \log n)$.

Tulos on tiukka ja tarkentaa edellistä tulosta tekijällä $\log n$. Aakkoston kokorajoituksesta seuraa esimerkiksi, että tulos pätee vakioaakkostolle, jos $S(n) = \Theta(n)$ bittiä. Toisaalta aakkoston koon on oltava vähintään $2^{\Omega(\sqrt{\log n})}$, jos $S(n) = \Theta(n\sqrt{\log n})$ bittiä.

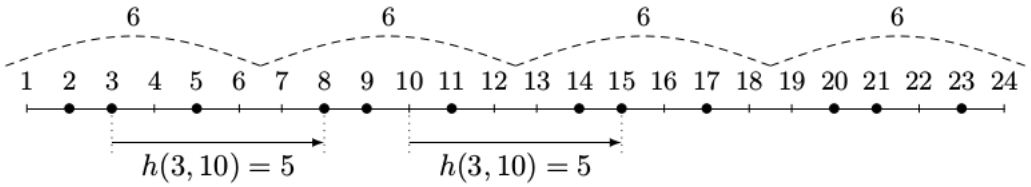
Tulokseen liittyy avoimia kysymyksiä. Ilman oletusta $S(n) = \Omega(n)$ bittiä esimerkiksi $S(n) = o(n)$ bittiä on mahdollinen. Tällöin saattaa päteä $S(n)T(n) = \Omega(n \log S(n))$, mutta tätä ei ole todistettu. Toiseksi tulos ei päde pienellä aakkostolla, jos konesanaan tallennetaan useita merkkejä, joista jokainen vie $\log_\sigma n$ bittiä, ja merkit voidaan lukea yhdellä käskyllä. Kolmanneksi tulos koskee deterministisiä algoritmeja mutta ei satunnaistettuja algoritmeja, jotka tuottavat oikean vastauksen $\texttt{lce}_S$-kyselyyn tietyllä todennäköisyydellä. Koska S on vain luettavissa, tulos ei päde algoritmeille, jotka muokkaavat merkkijonoa S sopivan tietorakenteen toteuttamiseksi. Avoimet kysymykset eivät ole pelkästään teoreettisia, sillä on olemassa näitä seikkoja hyödyntäviä algoritmeja $\texttt{lce}_S$-kyselyn toteuttamiseen.

Tarkastellaan esimerkkejä LCE-ongelman tilaa säästävistä toteutusajatuksista. Tyypillinen tapa luoda aika- ja tilavaativuuden uusia yhdistelmiä on tallentaa syötteestä jollain tavalla poimittu otos, joka lisää tilavaativuutta mutta parantaa aikavaativuutta. Ensimmäisessä esimerkissä käytetään t-peitettä.

Olkoot t positiivinen kokonaisluku ja joukko $Z = \{0, 1, \ldots, t-1\}$. Nyt joukko $D \subseteq Z$ on t-erotuspeite [50], jos $Z = \{(x - y) \mod t \mid x, y \in D\}$. Toisin sanoen jokainen joukon Z alkio voidaan ilmaista joukon D kahden alkion erotuksena mod t. Esimerkiksi $D = \{2, 3, 5\}$ on 6-erotuspeite, koska jakojäännösaritmetiikassa $2 - 2 = 0$, $3 - 2 = 1$, $5 - 3 = 2$, $5 - 2 = 3$, $3 - 5 = 4$ ja $2 - 3 = 5$.

Joukko $S(t) \subseteq \{1, 2, \ldots, n\}$ on kokonaislukujen $1, 2, \ldots, n$ t-peite, jos pätee kaksi ehtoa:

1. $S(t) = \{i \in \{1, 2, \ldots, n\} \mid (i \mod t) \in D\}$, missä D on jokin t-erotuspeite.
2. $h(i, j)$ on vakioaikainen funktio, jolle kaikilla $1 \leq i, j \leq n - t$ pätee $0 \leq h(i, j) < t$ ja $i + h(i, j),\ j + h(i, j) \in S(t)$.



Kuva 9: Esimerkki [19] 6-peitteestä $S(6) = \{2, 3, 5, 8, 9, 11, 14, 15, 17, 20, 21, 23\}$ kokonaisluvuille $1, 2, \ldots, 24$, kun $D = \{2, 3, 5\}$. Peitteen alkiot on merkitty janalle, ja ne toistuvat kuuden jaksoissa. Nyt pätee esimerkiksi $h(3, 10) = 5$, koska $3 + 5, 10 + 5 \in S(6)$.

Kuvassa 9 on esimerkki t-peitteestä.

Koska t-erotuspeite D voi kattaa enintään $|D|^2$ kokonaislukua, joukon D koko on vähintään $\sqrt{t}$. Määritelmän perusteella $S(t)$ sisältää arvon t pituisia jaksoja, joihin kuuluvat alkiot vastaavat joukon D alkioita. On osoitettu [9] jokaiselle $t \leq n$ olevan olemassa t-peite, jonka koko on $O(n/\sqrt{t})$ konesanaa ja jonka voi laskea ajassa $O(n/\sqrt{t})$.

Poimitaan merkkijonon $S[1..n]$ loppuosista otos [8]. Loppuosa $S[i..n]$ kuuluu otokseen, jos ja vain jos $i \in S(t)$. Esimerkiksi kuvan 9 mukaisella 6-peitteellä otokseen kuuluvat loppuosat $S[2..n]$, $S[3..n]$, $S[5..n]$, $S[8..n]$ ja niin edelleen. Otoksessa on $O(n/\sqrt{t})$ loppuosaa.

Otoksen loppuosista luodaan loppuosapuu T. Se vie tilaa $O(n/\sqrt{t})$ konesanaa. Kukin loppuosa lisätään puuhun ajassa $O(n)$, joten luominen vie aikaa $O(n^2/\sqrt{t})$. Puuta täydennetään apurakenteella, jonka avulla kahden lehden alin yhteinen esi-isä löytyy vakioajassa kyselyllä $\texttt{lca}_T$.

Kyselyyn $\texttt{lce}_S(i, j)$ vastaamiseksi verrataan loppuosia $S[i..n]$ ja $S[j..n]$ merkki kerrallaan, kunnes löytyy eroava merkki tai saavutaan merkkeihin $S[i + h(i, j)]$ ja $S[j + h(i, j)]$. Näistä merkeistä alkavat loppuosat ovat loppuosapuussa. Jos löytyi eroava merkki, palautetaan vastaus. Muuten etsitään puun avulla loppuosien pisimmän yhteisen alkuosan pituus l, ja vastaus $\texttt{lce}_S(i, j) = h(i, j) + l$. Koska $h(i, j) < t$ ja kysely $\texttt{lca}_T$ on vakioaikainen, kyselyn $\texttt{lce}_S(i, j)$ aikavaativuus on $O(t)$.

Tarkastellaan toisenlaista otosta käyttävää LCE-ongelman ratkaisua [8]. Merkkijonolle $S[1..n]$ lasketaan Karp–Rabin-tunnisteita. Olkoon alkuluku $2n^{c+4} < p \leq 4n^{c+4}$, missä vakio $c > 0$. Valitaan satunnaisesti $b \in \{0, 1, \ldots, p - 1\}$. Mille tahansa merkkijonon S osajonolle T voidaan laskea Karp–Rabin-tunniste kaavalla

$$\phi(T) = \sum_{k=1}^{|T|} T[k] \cdot b^k \bmod p.$$

Jos kaksi merkkijonon S osajonoa ovat samat, myös niiden tunnisteet ovat samat. Toisaalta kaava on valittu siten, että kahden eri osajonon tunnisteet ovat erisuuret suurella todennäköisyydellä. Tämä tarkoittaa vähintään todennäköisyyttä $1 - n^{-c}$. Siis kahden osajonon tunnisteita vertaamalla voidaan vakioajassa selvittää suurella

todennäköisyydellä, ovatko osajonot samat.

Olkoon $1 \leq t \leq n$. Lasketaan ja tallennetaan tunniste jokaiselle loppuosalle $S[kt..n]$, missä $1 \leq k \leq n/t$. Tunnisteita on $O(n/t)$ kappaletta. Ne vievät tilaa $O(n/t)$ konesanaa, jos jokainen tunniste vie vakiotilan. Tunnisteet voi laskea ajassa $O(n)$ käymällä S kerran läpi alusta loppuun.

Tallennettujen tunnisteiden avulla minkä tahansa osajonon $S[i..i+\alpha-1]$ tunnisteen voi laskea ajassa $O(t)$. Olkoot $k_1 = \lceil i/t \rceil$ ja $k_2 = \lceil (i+\alpha)/t \rceil$. Nyt osajonojen $S[k_1t..n]$ ja $S[k_2t..n]$ tunnisteet ovat jo olemassa. Niiden avulla saadaan vakioajassa

$$\phi(S[k_1t..k_2t-1]) = \phi(S[k_1t..n]) - \phi(S[k_2t..n]) \cdot b^{(k_2-k_1)t} \bmod p.$$

Jos i ja α ovat jaollisia arvolla t, vastaus löytyi. Laskenta on tällöin vakioaikainen. Muuten saatu tunniste vastaa sellaisen osajonon tunnistetta, joka eroaa pituudeltaan tai sijainniltaan enintään t merkkiä osajonon $S[i..i+\alpha-1]$ pituudesta tai sijainnista. Olemassa olevien tietojen avulla tunniste voidaan muuntaa osajonon $S[i..i+\alpha-1]$ tunnisteeksi vakioaikaisilla laskutoimituksilla, joita on enintään $O(t)$ kappaletta.

Kyselyyn $\texttt{lce}_S(i,j)$ vastataan palauttamalla se kyselyksi $\texttt{lce}_S(it,jt)$ tai kyselyksi $\texttt{lce}_S(it,jt+\gamma)$, missä $0 < \gamma < t$. Palautus vaatii enintään $O(t)$ vakioaikaista laskutoimitusta tunnisteilla. Katsotaan esimerkkinä kyselyn $\texttt{lce}_S(it,jt)$ toteutusajatusta. Määritellään $\phi_k^l = \phi(T[kt..(k+2^l)t-1])$, missä kokonaisluku $l \geq 0$. Nyt ϕ_k^l voidaan laskea vakioajassa kaikilla k ja l. Ensin etsitään suurin l, jolla pätee $\phi_i^l = \phi_j^l$. Tällöin loppuosilla $S[i..n]$ ja $S[j..n]$ on yhteinen alkuosa, jonka pituus on vähintään 2^lt. Seuraavaksi etsitään suurin l kyselylle $\texttt{lce}_S((i+2^l)t,(j+2^l)t)$. Iterointia jatketaan, kunnes arvoa l ei enää löydy. Tällöin pisimmän yhteisen alkuosan vielä tutkimattoman loppuosan pituus selvitetään vertaamalla loppuosia merkki kerrallaan.

Kyselyn $\texttt{lce}_S(i,j)$ aikavaativuudeksi tulee $O(t \log(\texttt{lce}_S(i,j)/t))$. Tunnisteiden laskentatavan vuoksi vastaus on oikea suurella todennäköisyydellä. Jotta vastaus on varmuudella oikea, kahden loppuosan tunnisteiden on oltava samat, jos ja vain jos loppuosat ovat samat. Se on mahdollista varmistaa tunnisteita laskettaessa. Tällöin esikäsittelyn aikavaativuus kasvaa luokasta $O(n)$ luokkaan $O(n \log n)$ siten, että aikavaativuus kuuluu uuteen luokkaan suurella todennäköisyydellä.

Edellä kuvatut t-peitteeseen ja Karp–Rabin-tunnisteisiin perustuvat LCE-ongelman ratkaisut käyttävät systemaattista tietorakennetta. Ne tarvitsevat kyselyihin alkuperäistä merkkijonoa $S[1..n]$, joka vie tilaa $O(n \log \sigma)$ bittiä, missä σ on aakkoston koko. Merkkijonon S tilantarve ei ole mukana edellä mainituissa tilavaativuuksissa.

LCE-ongelmaan on myös kehitetty [44] Karp–Rabin-tunnisteita käyttävä ratkaisu, jossa S korvataan tietorakenteella. Ratkaisu vie tilaa vakiomäärän konesanoja enemmän kuin S ja palauttaa oikean vastauksen suurella todennäköisyydellä. Vastaamiseen ei tarvita merkkijonoa S. Sen voi kuitenkin palauttaa tietorakenteesta, joten informaatio ei vähene rakennetta luotaessa.

LCE-ongelmaan on olemassa [27] tiivistämiseen perustuvia ratkaisuja. Niissä S esitetään kielioppina, joka tuottaa ainoastaan merkkijonon S. Esikäsittelyn jälkeen alkuperäistä merkkijonoa ei tarvita. Jos S on hyvin tiivistyvä, tietorakenteen koko

voi olla alilineaarinen merkkijonoon S verrattuna. Näihin ratkaisuihin ei syvennytä tarkemmin.

5 Koodaava tietorakenne ongelman ratkaisussa

Luvussa 4 tarkasteltiin tilaa säästävien tietorakenteiden yleisiä toteutustapoja, LCE-ongelman aika–tila-vaihtokauppaa ja esimerkkejä ongelman tilaa säästävistä ratkaisuista. Osassa ratkaisuja käytettiin systemaattisia tietorakenteita, jotka tarvitsevat alkuperäisen merkkijonon `lce`-kyselyä varten. Toinen käyttökelpoinen tapa luoda tarvittava tietorakenne perustuu tiivistämiseen. Hyvin tiivistyvästä merkkijonosta voi luoda kyselyyn vastaamiseen riittävän tietorakenteen, jonka koko on pienimmillään alilineaarinen merkkijonoon verrattuna.

Tässä luvussa tutustutaan yksityiskohtaisesti LCE-ongelman vakioaikaiseen ratkaisuun, joka ei käytä alkuperäistä merkkijonoa tietorakenteen luomisen jälkeen. Ratkaisun tilavaativuus on riittävän pienellä aakkostolla alilineaarinen merkkijonoon nähden, vaikka merkkijono ei olisi hyvin tiivistyvä Lempel–Ziv 77 -tekijöihinjaon avulla. Ratkaisussa käytetään hyväksi useita edellisissä luvuissa esiteltyjä tietorakenteita.

5.1 Algoritmin yleiskuvaus

Käydään yleisesti läpi algoritmi $\texttt{general\_lce}_S(i, j)$. Se palauttaa järjestetyn aakkoston $[1..\sigma]$ merkkijonon $S[1..n]$ loppuosien $S[i..n]$ ja $S[j..n]$ pisimmän yhteisen alkuosan pituuden eli arvon $\texttt{lce}_S(i, j)$. Algoritmissa [50] käytetään apuna kyselyä

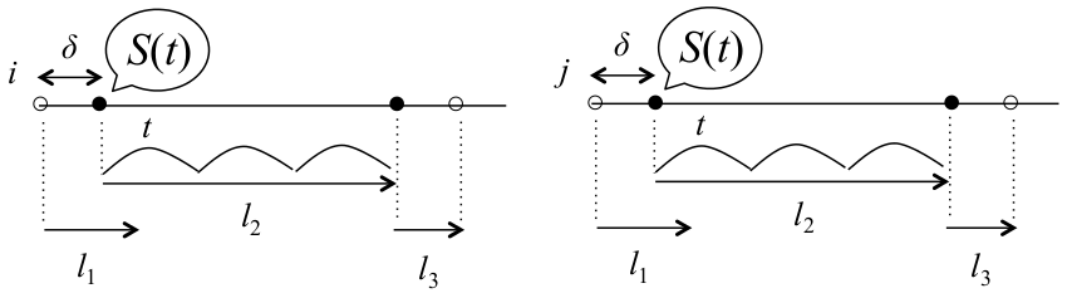
$$\texttt{short\_lce}_S(i, j) = \min(\texttt{lce}_S(i, j), t),$$

missä t on kokonaisluvuille $1, 2, \ldots, n$ lasketun t-peitteen $S(t)$ arvo t. Kysely palauttaa siis enintään arvon t. Määritellään arvoille $i, j \in S(t)$ myös kysely

$$\texttt{long\_lce}_S(i, j) = \lfloor \texttt{lce}_S(i, j)/t \rfloor.$$

Siten tämä apukysely palauttaa tiedon, montako yhteistä arvon t pituista osajonoa on loppuosien $S[i..n]$ ja $S[j..n]$ alussa.

Kysely $\texttt{general\_lce}_S(i, j)$ toteutetaan kolmessa vaiheessa, jotka esitetään algoritmissa 2 [19] ja kuvassa 10. Ensin suoritetaan kysely $\texttt{short\_lce}_S(i, j)$. Jos sen paluuarvo on pienempi kuin t, löytyi arvo $\texttt{lce}_S(i, j)$, ja algoritmin suoritus päättyy. Muutoin paluuarvo on t, josta päätellään, että $\texttt{lce}_S(i, j)$ on vähintään t. Jatketaan toiseen vaiheeseen käyttämällä t-peitteen funktiota $h(i, j)$, jonka paluuarvo tallennetaan muuttujaan δ. Nyt pätee $i + \delta$, $j + \delta \in S(t)$, missä $0 \leq \delta < t$. Seuraavaksi kyselyn $\texttt{long\_lce}_S(i + \delta, j + \delta)$ paluuarvo kertoo, montako täyttä arvon t pituista jaksoa etsittävä pisin yhteinen alkuosa jatkuu eteenpäin merkeistä $S[i+\delta]$ ja $S[j+\delta]$. Muunnetaan paluuarvo merkkien määräksi kertomalla se arvolla t, ja tallennetaan



Kuva 10: Algoritmin $\texttt{general\_lce}_S(i, j)$ kolmivaiheinen toimintaperiaate [50].

vastaus muuttujaan l_2. Kolmannessa vaiheessa tehdään taas kysely $\texttt{short\_lce}_S$. Se palauttaa etsittävän alkuosan vielä selvittämättömän, alle arvon t pituisen loppuosan pituuden. Lopuksi palautetaan vastauksena apumuuttujiin tallennettujen pituuksien summa.

Algoritmi 2 $\texttt{general\_lce}_S(i, j)$

Syöte: merkkijonon S loppuosien $S[i..n]$ ja $S[j..n]$ alkukohtien indeksit i ja j
Tuloste: $\texttt{lce}_S(i, j)$ eli loppuosien pisimmän yhteisen alkuosan pituus

1: $l_1 \leftarrow \texttt{short\_lce}_S(i, j)$ ▷ Vaihe 1
2: **if** $l_1 < t$ **then**
3: **return** l_1
4: **end if**
5: $\delta \leftarrow h(i, j)$ ▷ Vaihe 2
6: $l_2 \leftarrow t \cdot \texttt{long\_lce}_S(i + \delta, j + \delta)$
7: $l_3 \leftarrow \texttt{short\_lce}_S(i + \delta + l_2, j + \delta + l_2)$ ▷ Vaihe 3
8: **return** $\delta + l_2 + l_3$

5.2 Kysely `short_lce`

Olkoot $S[1..n]$ merkkijono, jonka viimeinen merkki \$ ei esiinny muualla merkkijonossa, ja kokonaisluku $1 \leq q \leq n$. Määritellään [50] apukäsitteeksi joukko $Subs_q(S) = \{S[i.. \min(i + q - 1, n)] \mid 1 \leq i \leq n\}$. Siis joukkoon kuuluvat arvon q pituiset merkkijonon S osajonot ja osajonon $S[n - q + 1..n]$ loppuosat. Esimerkiksi merkkijonolle $S[1..6] =$ `ananas` joukko $Subs_3(S) = \{\texttt{ana}, \texttt{nan}, \texttt{nas}, \texttt{as}, \texttt{s}\}$. Olkoon merkkijonon S q-typistetty loppuosapuu (*q-truncated suffix tree*) q-$TST(S)$ joukon $Subs_q(S)$ alkioista luotu loppuosapuu. Puussa jokainen polku juuresta lehteen vastaa enintään arvon q pituista osajonoa. Olkoon $l(i)$ lehti, jonne päättyvä polku vastaa merkistä $S[i]$ alkavaa osajonoa. Jos $q = n$, niin q-$TST(S)$ ja merkkijonon S tavallinen loppuosapuu ovat samat.

Kyselyn $\texttt{short\_lce}_S(i, j)$ toteutusajatuksena [50] on luoda puu $2t$-$TST(S)$, missä arvo t saadaan kokonaisluvuille $1, 2, \ldots, n$ lasketusta t-peitteestä, ja etsiä puusta lehtien $l(i)$ ja $l(j)$ alin yhteinen esi-isä. Tästä solmusta selviää juuresta siihen tulevaa

polkua vastaavan merkkijonon pituus. Vakioaikainen, tilaa säästävä toteutus ei ole aivan yksinkertainen.

Toteutuksessa tarvitaan nopeaa pääsyä puun $2t$-$TST(S)$ lehtiin. Se järjestetään osoittimilla, mutta tilan säästämiseksi tallennetaan vain noin n/t osoitinta. Olkoon otoskohtien joukko $Q_n = \{1 + kt \mid 0 \leq k \leq \lceil n/t \rceil - 1\}$. Nyt jokaista $j \in Q_n$ kohti tallennetaan osoitin, joka viittaa lehteen $l(j)$.

Merkitään $\alpha(i) = \max\{j \in Q_n \mid j \leq i\}$. Eli $\alpha(i)$ on lähin arvoa i edeltävä otoskohta tai, jos $i \in Q_n$, niin $\alpha(i) = i$. Mille tahansa $1 \leq i \leq n$ pätee $\alpha(i) = 1 + \lfloor \frac{i-1}{t} \rfloor t$. Näin ollen puun $2t$-$TST(S)$ lehti $l(\alpha(i))$ löydetään tallennetun osoittimen avulla vakioajassa.

Seuraavaksi toteutetaan vakioaikainen pääsy mihin tahansa puun $2t$-$TST(S)$ lehteen $l(i)$. Tätä varten luodaan ensin suunnattu verkko $G = (V, E)$. Solmujen joukko $V = Subs_{2t}(S)$, eli jokainen solmu vastaa yhtä osajonoa. Olkoon v_i solmu, joka vastaa merkistä $S[i]$ alkavaa osajonoa. Joukko $E = \{(v_i, S[i-1], v_{i-1}) \mid 1 < i \leq n\}$, missä $(v_i, S[i-1], v_{i-1})$ on merkillä $S[i-1]$ nimetty kaari solmusta v_i solmuun v_{i-1}.

Suunnatussa verkossa G on polku $v_i \rightsquigarrow v_{\alpha(i)}$ pituudeltaan $d = i - \alpha(i)$. Ei kuitenkaan ole nopeaa tapaa liikkua helposti toiseen suuntaan solmusta $v_{\alpha(i)}$ solmuun v_i. Tämä ongelma ratkaistaan luomalla verkon G virittävä puu T. Sen juuri on solmu v_n, joka vastaa merkkijonon S viimeistä merkkiä \$.

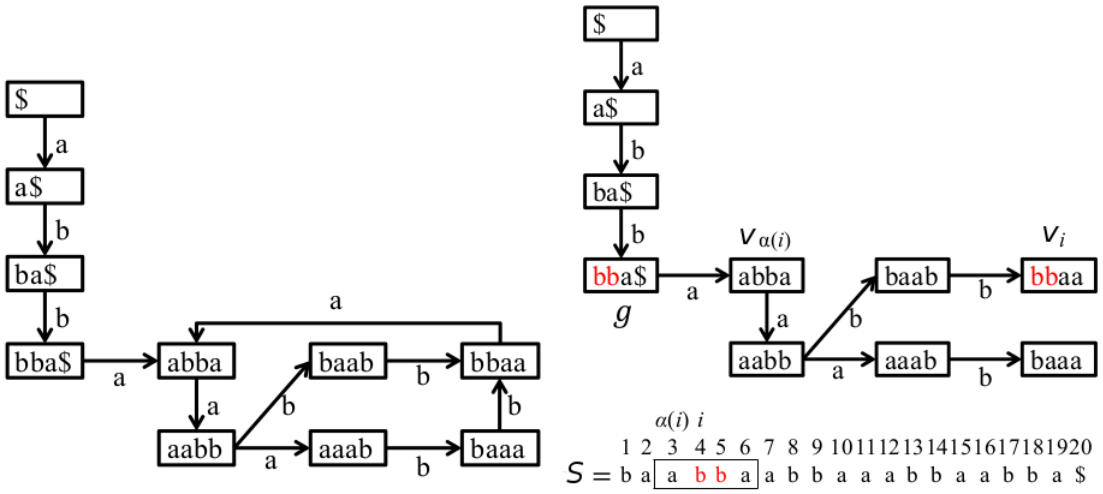
Millä tahansa verkon G virittävällä puulla T on kaksi olennaista ominaisuutta:

1. Juuresta alkava polku, jonka pituus on $2t$, ei haaraudu.
2. Oletetaan, että $1 \leq i < n - 2t$ ja $0 \leq d < t$. Olkoot solmu g solmun $v_{\alpha(i)}$ esi-isä ja d polun $g \rightsquigarrow v_{\alpha(i)}$ pituus. Tällöin solmua g vastaavan osajonon ja solmua v_i vastaavan osajonon t ensimmäistä merkkiä ovat samat.

Ensimmäinen ominaisuus seuraa siitä, että merkkijono S päättyy muualla esiintymättömään merkkiin \$ ja että puun juuri vastaa osajonoa $S[n..n] = \$$. Toisen ominaisuuden todistamiseksi huomataan ensimmäisestä ominaisuudesta ja epäyhtälöstä $\alpha(i) \leq i < n - 2t$ seuraavan, että solmu $v_{\alpha(i)}$ on puussa T vähintään syvyydellä $2t$. Siis solmua vastaavan osajonon pituus on $2t$. Koska $d < t$, niin solmua $v_{\alpha(i)}$ vastaavan osajonon merkit välillä $[d + 1..d + t]$ ovat samat kuin solmua v_i vastaavan osajonon t ensimmäistä merkkiä. Jos puussa T kuljetaan solmusta $v_{\alpha(i)}$ kaarten vastaisesti kohti esi-isää g, jokaisella askeleella poistetaan solmua $v_{\alpha(i)}$ vastaavasta osajonosta ensimmäinen merkki. Näin ollen solmua g vastaavan osajonon ja solmua v_i vastaavan osajonon t ensimmäistä merkkiä ovat samat.

Kuvassa 11 on esimerkki verkosta G ja virittävästä puusta T.

Puuta T täydennetään tietorakenteella, jonka avulla kysely $\texttt{level\_ancestor}_T(u, s)$ palauttaa vakioajassa solmun v, missä v on solmun u syvyydellä s puussa oleva esi-isä [4]. Solmun v syvyys, $depth(v)$, tarkoittaa kaarten määrää lyhimmällä polulla juuresta solmuun v. Näin ollen virittävässä puussa T solmun $v_{\alpha(i)}$ esi-isä g saadaan kyselyllä $\texttt{level\_ancestor}_T(v_{\alpha(i)}, depth(v_{\alpha(i)}) - d)$.



Kuva 11: Vasemmalla on esimerkki [50] merkkijonoa $S[1..20]$ vastaavasta suunnatusta verkosta G arvolla $t = 2$. Solmut ovat joukko $V = Subs_4(S)$. Oikealla on verkon G virittävä puu T. Olkoon $i = 4$. Nyt $\alpha(i) = 1 + \lfloor \frac{i-1}{t} \rfloor t = 3$, joten $d = i - \alpha(i) = 1$. Solmu g on siten solmun $v_{\alpha(i)}$ ensimmäinen esi-isä puussa T. Solmujen g ja v_i osajonojen kaksi ensimmäistä merkkiä ovat samat.

Puun $2t$-$TST(S)$ lehtien ja verkon G solmujen välillä on bijektiivinen vastaavuus, jossa lehti $l(i)$ vastaa solmua v_i. Verkossa G ja virittävässä puussa T puolestaan on samat solmut. Siten liikkuminen puiden $2t$-$TST(S)$ ja T vastinsolmujen välillä on mahdollista toteuttaa vakioajassa ilman lisärakenteita. Näin puun $2t$-$TST(S)$ lehtiin osoittavien osoittimien avulla löydetään myös puun T vastaavat solmut.

Edellä esiteltyjen tietorakenteiden avulla kyselyn $\texttt{short\_lce}_S(i, j)$ voi toteuttaa [50] algoritmin 3 mukaisesti. Riveillä 1 ja 2 lasketaan parametreja i ja j vastaavat otoskohdat. Riveillä 3 ja 4 etsitään tallennettujen osoittimien avulla virittävän puun T solmut $v_{\alpha(i)}$ ja $v_{\alpha(j)}$. Riveillä 5 ja 6 tallennetaan apumuuttujiin d ja d' tiedot siitä, kuinka monennet solmujen esi-isät halutaan löytää. Riveillä 7 ja 8 etsitään solmujen $v_{\alpha(i)}$ ja $v_{\alpha(j)}$ halutut esi-isät puussa T kyselyn $\texttt{level\_ancestor}_T$ avulla. Rivillä 9 kysely $\texttt{lca}_{2t\text{-}TST(S)}$ palauttaa esi-isiä vastaavien puun $2t$-$TST(S)$ lehtien alimman yhteisen esi-isän, joka tallennetaan muuttujaan x. Rivillä 10 $str(x)$ tarkoittaa juuresta solmuun x kulkevaa polkua vastaavaa merkkijonoa, joten algoritmi palauttaa tämän merkkijonon pituudesta ja arvosta t pienemmän.

5.3 Kysely `long_lce`

Kysely $\texttt{long\_lce}_S(i, j)$ toteutetaan [50] luomalla merkkijonosta $S[1..n]$ merkkijono $code(S)$, jonka avulla kyselyyn vastataan. Olkoot $S(t)$ kokonaisluvuille $1, 2, \ldots, n$ laskettu t-peite ja D siihen liittyvä t-erotuspeite. Jos $i \in S(t)$, kutsutaan osajonoa $S[i..\min(i + t - 1, n)]$ t-lohkoksi. Kysely on määritelty vain arvoilla $i, j \in S(t)$ ja palauttaa yhteisten arvon t pituisten osajonojen määrän loppuosien $S[i..n]$ ja $S[j..n]$

Algoritmi 3 $\texttt{short\_lce}_S(i, j)$

Syöte: merkkijonon S loppuosien $S[i..n]$ ja $S[j..n]$ alkukohtien indeksit i ja j
Tuloste: arvon $\texttt{lce}_S(i, j)$ ja peitteen $S(t)$ arvon t minimi

1: $\alpha(i) \leftarrow 1 + \lfloor \frac{i-1}{t} \rfloor t$ ▷ Lasketaan otoskohdat $\alpha(i)$ ja $\alpha(j)$
2: $\alpha(j) \leftarrow 1 + \lfloor \frac{j-1}{t} \rfloor t$
3: $v_{\alpha(i)} \leftarrow p(\alpha(i))$ ▷ Etsitään puun T solmut $v_{\alpha(i)}$ ja $v_{\alpha(j)}$
4: $v_{\alpha(j)} \leftarrow p(\alpha(j))$
5: $d \leftarrow i - \alpha(i)$ ▷ Etsitään solmujen esi-isät
6: $d' \leftarrow j - \alpha(j)$
7: $v_{\alpha(i)-d} \leftarrow \texttt{level\_ancestor}_T(v_{\alpha(i)}, depth(v_{\alpha(i)}) - d)$
8: $v_{\alpha(j)-d'} \leftarrow \texttt{level\_ancestor}_T(v_{\alpha(j)}, depth(v_{\alpha(j)}) - d')$
9: $x \leftarrow \texttt{lca}_{2t\text{-}TST(S)}(l(\alpha(i) - d), l(\alpha(j) - d'))$
10: **return** $\min(|str(x)|, t)$

alussa. Jokaista t-lohkoa voi siksi käsitellä kyselyä toteutettaessa yhtenä merkkinä.

Merkkijono $code(S)$ luodaan neljässä vaiheessa:

1. Merkkijonon S t-lohkot järjestetään aakkosjärjestykseen. Lohkoista muodostetaan joukko poistamalla kaksoiskappaleet.

2. Järjestetyt t-lohkot numeroidaan peräkkäisillä luonnollisilla luvuilla siten, että järjestyksessä ensimmäisen lohkon arvo on 1. Olkoon r_i merkistä $S[i]$ alkavan t-lohkon arvo. Arvot muodostavat kokonaislukuaakkoston.

3. Jokaista $i \in D$ kohti luodaan t-lohkojen arvoja katenoimalla merkkijono $z_i = r_i r_{i+t} r_{i+2t} \dots r_{i+mt}$, missä $m = \lfloor \frac{n-i-1}{t} \rfloor$. Merkkijonojen määrä $k = |D|$. Näistä merkkijonoista katenoidaan merkkijono $code(S) = z_{i_1}\$_{i_1} z_{i_2}\$_{i_2} \dots z_{i_k}\$_{i_k}$. Merkit $\$_{i_1}, \$_{i_2}, \dots, \$_{i_k}$ valitaan siten, että kukin niistä esiintyy merkkijonossa $code(S)$ täsmälleen kerran.

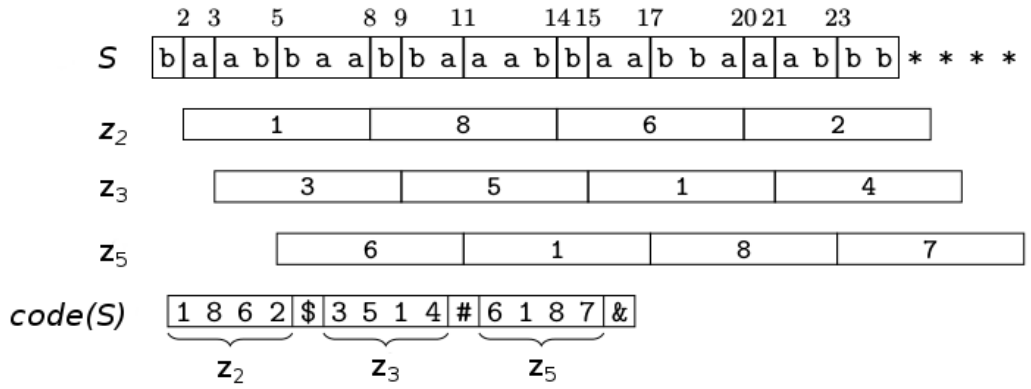
4. Merkkijonoa $code(S)$ täydennetään kyselyn $\texttt{lce}_{code(S)}(i, j)$ vakioaikaiseksi tekevillä apurakenteilla. Nämä ovat käänteinen loppuosataulukko SA^{-1}, LCP-taulukko ja osavälin pienimmän alkion indeksin palauttavan kyselyn $\texttt{rmq}_{LCP}$ toteutus, jotka esiteltiin luvussa 2.2.

Alkuperäisen merkkijonon kohtaa $S[i]$, missä $i \in S(t)$, vastaa kohta $code(S)[i']$. Se lasketaan kaavalla $i' = |z_1\$_1 z_2\$_2 \dots z_{x-1}\$_{x-1}| + \frac{i-x}{t} + 1$, missä $x = i \mod t$.

Nyt kysely $\texttt{long\_lce}_S(i, j)$ palautuu kyselyksi $\texttt{lce}_{code(S)}(i', j')$, koska merkkijonossa $code(S)$ yksi merkki vastaa merkkijonon S yhtä t-lohkoa. Kuvassa 12 on esimerkki merkkijonon $code(S)$ luomisesta ja käytöstä.

5.4 Tietorakenteen aika- ja tilavaativuudet

Tarkastellaan kyselyn $\texttt{general\_lce}_S(i, j)$ tietorakenteiden aika- ja tilavaativuuksia järjestetyn aakkoston $[1..\sigma]$ merkkijonolle $S[1..n]$. Kyselyä kokonaisuutena käsitel-



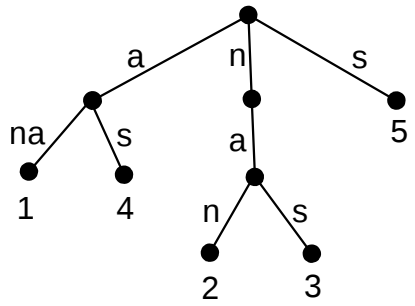
Kuva 12: Esimerkki [19] merkkijonon $code(S)$ luomisesta merkkijonosta $S[1..24]$, kun $t = 6$ ja $D = \{2, 3, 5\}$. Ylhäällä näkyvät 6-peitteen alkiot ja lohkoissa aakkosjärjestyksen mukaiset arvot. $code(S)$ saadaan katenoimalla merkkijonot z_2, z_3 ja z_5 lisäten samalla näistä jokaisen loppuun muualla merkkijonossa esiintymätön merkki. Nyt $|code(S)| = 15$. Olkoot esimerkiksi $i = 2$ ja $j = 17$, jolloin $x_i = i \mod t = 2$ ja $x_j = j \mod t = 5$. Saadaan $i' = |z_1\$_1| + \frac{i-x_i}{t} + 1 = 0 + \frac{2-2}{6} + 1 = 0 + 0 + 1 = 1$ ja $j' = |z_1\$_1 z_2\$_2 z_3\$_3 z_4\$_4| + \frac{j-x_j}{t} + 1 = |z_2\$z_3\#| + \frac{17-5}{6} + 1 = 10 + 2 + 1 = 13$. Näin ollen kysely $\texttt{long\_lce}_S(2, 17)$ palautuu kyselyksi $\texttt{lce}_{code(S)}(1, 13)$.

lään seuraavassa luvussa.

Kun $t \leq n$, kokonaisluvuille $1, 2, \ldots, n$ laskettu t-peite vie tilaa $O(n/\sqrt{t})$ konesanaa, ja sen voi laskea ajassa $O(n/\sqrt{t})$ [9]. Funktio $h(i, j)$ on määritelmänsä mukaan vakioaikainen.

Puulle q-$TST(S)$ on olemassa [52] merkkijonosta S johdettu merkkijono S' pituudeltaan $O(|Subs_q(S)|)$. S' riittää kaarten viitteitä varten, eikä tällöin tarvita merkkijonoa S. Merkkijonon S' luomiseksi asetetaan $S' = S[1..q]$. Sitten tarkistetaan kohdista $S[i]$ alkavat arvon q pituiset osajonot, missä i saa arvot $2, 3, \ldots, n - q + 1$. Jos i on osajonon ensimmäisen esiintymän alkukohta merkkijonossa S, merkkijonon S' loppuun lisätään osajonon viimeinen merkki. Kuvassa 13 on esimerkki. S' ei välttämättä ole lyhin kaarten viitteisiin riittävä merkkijono, mutta sen voi luoda yksinkertaisesti. S' ja q-$TST(S)$ voidaan luoda ajassa $O(n \log \sigma)$, ja ne mahtuvat $O(|Subs_q(S)|)$ konesanaan. Luomisen aikainen tilavaativuus on $O(|Subs_q(S)|)$ konesanaa. Käytännössä merkkijonojen S ja S' kokojen erotus on samaa suuruusluokkaa kuin merkkijonon S tavallisen loppuosapuun ja puun q-$TST(S)$ kokojen erotus.

Puu q-$TST(S)$ ja merkkijono S' mahtuvat myös $O(zq)$ konesanaan [50], missä z on merkkijonon S Lempel–Ziv 77 -tekijöiden määrä. Tämä nähdään osoittamalla, että $|Subs_q(S)| = O(zq)$. Oletetaan ensin, että $q = 1$. Nyt joukko $Subs_1(S)$ sisältää pelkkiä aakkoston merkkejä. Siis selvästi $|Subs_1(S)| \leq z$, joten väite pätee. Oletetaan sitten, että $q \geq 2$. Olkoon $p \in Subs_q(S)$ osajono pituudeltaan q. Nyt osajonon p ensimmäinen esiintymä merkkijonossa S kattaa vähintään kahta peräkkäistä Lempel–Ziv 77 -tekijää f_k ja f_{k+1}. Jos p kattaisi vain yhtä tekijää, tekijöihinjaon määritelmän nojalla kyseessä ei olisi osajonon p ensimmäinen esiintymä. Tällaisia



Kuva 13: Merkkijonon $S[1..6] =$ `ananas` loppuosapuu 3-$TST(S)$. Puun polut juuresta lehtiin vastaavat joukon $Subs_3(S)$ alkioita. Merkkijonon kolmen pituiset osajonot ovat `ana`, `nan`, `ana` ja `nas`. Näistä luotu merkkijono $S' =$ `anans` riittää kaarten merkitsemiseen. Loppuosapuu 3-$TST(S')$ ei kuitenkaan olisi sama kuin kuvassa, koska esimerkiksi `ans` on merkkijonon S' mutta ei merkkijonon S osajono.

kumpaakin tekijää f_k ja f_{k+1} kattavia osajonoja pituudeltaan q voi olla korkeintaan $q-1$ kappaletta. Peräkkäisiä tekijäpareja puolestaan on $z-1$ kappaletta. Siis merkkijonon S erilaisia osajonoja pituudeltaan q on $O(zq)$ kappaletta. Joukko $Subs_q(S)$ sisältää myös $q-1$ osajonoa, jotka ovat lyhyempiä kuin q. Näin ollen pätee $|Subs_q(S)| = O(zq)$.

Siten puu $2t$-$TST(S)$ ja sen kaarten merkitsemiseen riittävä merkkijono S' voidaan luoda ajassa $O(n \log \sigma)$, luomisen tilavaativuus on $O(zt)$ konesanaa, ja puu ja S' mahtuvat $O(|Subs_{2t}(S)|) = O(zt)$ konesanaan. Jos $O(zt) > O(n)$, niin $2t$-$TST(S)$ kannattaa toteuttaa normaalina loppuosapuuna käyttäen merkkijonoa $S[1..n]$, jolloin tilavaativuudeksi tulee $O(n)$ konesanaa. Jatkossa oletetaan pätevän $zt < n$, koska vain nämä tapaukset ovat kiinnostavia tilansäästön kannalta.

Alimman yhteisen esi-isän etsintää eli kyselyä $\texttt{lca}_{2t\text{-}TST(S)}$ varten puuta täydennetään apurakenteella [3]. Sen avulla kysely on vakioaikainen. Apurakenne mahtuu $O(zt)$ konesanaan, ja sen luominen vie aikaa $O(zt)$ ja tilaa $O(zt)$ konesanaa.

Lisäksi tallennetaan noin n/t osoitinta, jotka viittaavat puun $2t$-$TST(S)$ lehtiin. Osoittimet vievät tilaa $O(n/t)$ konesanaa [50].

Verkko $G = (V, E)$ on de Bruijn -verkon muunnos. De Bruijn -verkko on suunnattu verkko, jonka solmujen joukon muodostavat merkkijonon $S[1..n]$ osajonot pituudeltaan q. Osajonoa $S[i..i+q-1]$ vastaavasta solmusta on kaari solmuun $S[i+1..i+q]$, missä $1 \leq i \leq n-q$. Siten kaarella yhdistetyillä solmuilla on yhteinen osajono pituudeltaan $q-1$. Merkkijonon S de Bruijn -verkon voi luoda olemassa olevan puun q-$TST(S)$ avulla. Tällöin luomisen aika- ja tilavaativuus on lineaarinen suhteessa valmiin verkon kokoon.

Verkkoa G varten luodaan ensin puun $2t$-$TST(S)$ avulla de Bruijn -verkko merkkijonon S osajonoista, joiden pituus on $2t$. Sen kaaret käännetään. Koska verkon G solmuja ovat kaikki joukon $Subs_{2t}(S)$ alkiot, verkko valmistuu lisäämällä solmuiksi puuttuvat $2t-1$ lyhyempää osajonoa ja niille kaaret.

Verkossa G on $|Subs_{2t}(S)| = O(zt)$ solmua. Jokainen kaari vastaa yhtä joukon

$Subs_{2t+1}(S)$ alkiota, eikä mitään alkiota kohti ole enempää kuin yksi kaari. Siis verkossa G on $O(|Subs_{2t+1}(S)|) = O(zt)$ kaarta. Näin ollen valmiin verkon G koko on $O(zt)$ konesanaa. Lineaarisuuden vuoksi verkon luomisen aikavaativuus on $O(zt)$ ja tilavaativuus $O(zt)$ konesanaa, sillä kaarten kääntäminen ja ylimääräisten solmujen ja kaarten lisääminen ei nosta vaativuuksia de Bruijn -verkon vaativuusluokista.

Virittävä puu T sisältää samat solmut kuin verkko G ja enintään saman määrän kaaria. Siis myös puun T tilavaativuus on $O(zt)$ konesanaa. Koska mikä tahansa virittävä puu kelpaa, puun T voi luoda lineaarisessa ajassa esimerkiksi käymällä G läpi syvyys- tai leveyssuunnassa alkaen osajonoa $S[n..n]$ vastaavasta solmusta ja poistamalla ylimääräiset kaaret. Puun T luomisen aikavaativuus on $O(zt)$ ja tilavaativuus $O(zt)$ konesanaa.

Vakioaikaista kyselyä $\texttt{level\_ancestor}_T$ varten puuta T täydennetään apurakenteella [4]. Apurakenne mahtuu $O(zt)$ konesanaan, ja sen luominen vie aikaa $O(zt)$ ja tilaa $O(zt)$ konesanaa.

t-lohkot voi järjestää yleiskäyttöistä järjestämisalgoritmia nopeammin [50] luomalla puu $t\text{-}TST(S)$ ja verkkoa G vastaava verkko, jonka solmuja ovat joukon $Subs_t(S)$ alkiot. Puun lehdet ja verkon solmut vastaavat toisiaan yksi yhteen. Koska puun lehdet ovat järjestyksessä, lehtiin voi merkitä järjestyksen käymällä puu läpi. Tämän jälkeen jokaiselle t-lohkolle saadaan järjestys sitä vastaavasta lehdestä. Puun ja verkon luominen ja läpikäynti vievät aikaa yhteensä $O(n \log \sigma)$. Toimien tilavaativuus on $O(zt)$ konesanaa.

Merkkijono $code(S) = z_{i_1}\$_{i_1}z_{i_2}\$_{i_2} \ldots z_{i_k}\$_{i_k}$ sisältää $|S(t)| = O(n/\sqrt{t})$ kokonaislukua ja $k = |D| = O(\sqrt{t})$ merkkiä $\$_i$. Jokainen z_i on kokonaislukuaakkoston $[1..|S(t)|] \subset [1..n]$ merkkijono. Merkkijonoa $code(S)$ voi siis pitää merkkijonona kokonaislukuaakkostossa kooltaan $O(n)$. Välillä $1 \leq t \leq n$ lausekkeen $(n/\sqrt{t}) + \sqrt{t}$ arvo on laskeva, joten lauseke saa suurimman arvonsa kohdassa $t = 1$. Täten $code(S)$ vie tilaa $O((n/\sqrt{t}) + \sqrt{t}) = O(n/\sqrt{t})$ konesanaa. Järjestettyjä t-lohkoja käyttäen sen voi luoda ajassa $O(n/\sqrt{t})$ ja tilassa $O(n/\sqrt{t})$ konesanaa.

Vakioaikaista kyselyä $\texttt{lce}_{code(S)}$ varten merkkijonoa $code(S)$ täydennetään käänteisellä loppuosataulukolla, LCP-taulukolla ja osavälin pienimmän alkion indeksin palauttavan kyselyn toteutuksella [3]. Apurakenteiden luomisen aika- ja tilavaativuudet ja käytön tilavaativuus ovat samat kuin merkkijonolla.

Taulukkoon 5 on koottu esillä olleiden tietorakenteiden aika- ja tilavaativuudet.

5.5 Kyselyjen aika- ja tilavaativuudet

Tarkastellaan kyselyn $\texttt{general\_lce}_S$ aika- ja tilavaativuuksia taulukon 5 tietojen avulla [50]. Taulukosta huomataan tarvittavien tietorakenteiden käytön olevan vakioaikaista.

Kysely $\texttt{short\_lce}_S$ palauttaa vastauksen ajassa $O(1)$. Algoritmin 3 (sivulla 39) rivien 1, 2, 5 ja 6 aritmeettiset operaatiot ovat selvästi vakioaikaisia. Riveillä 3 ja 4 käytetään valmiiksi tallennettuja osoittimia. Riveillä 7 ja 8 kyselyä $\texttt{level\_ancestor}_T$

Taulukko 5: Algoritmin $\texttt{general\_lce}_S$ tietorakenteiden aika- ja tilavaativuudet järjestetyn aakkoston $[1..\sigma]$ merkkijonolle $S[1..n]$. Taulukossa z on merkkijonon S Lempel–Ziv 77 -tekijöiden määrä ja $1 \leq t \leq n$. Lisäksi oletetaan, että $zt < n$. Tilavaativuudet on ilmaistu konesanoina. Puuhun $2t$-$TST(S)$, virittävään puuhun T ja merkkijonoon $code(S)$ liittyvät niiden yhteydessä mainitut apurakenteet, joilla operaatiot saadaan vakioaikaisiksi. Verkkoa G ei tarvita algoritmin suorituksessa, vaan se on välivaihe puun T luomisessa. Samoin t-lohkojen järjestäminen on välivaihe merkkijonon $code(S)$ luomisessa.

	Käyttö		Esikäsittely	
	Aikav.	Tilav.	Aikav.	Tilav.
t-peite $S(t)$	$O(1)$	$O(\frac{n}{\sqrt{t}})$	$O(\frac{n}{\sqrt{t}})$	$O(\frac{n}{\sqrt{t}})$
$2t$-$TST(S)$, $\texttt{lca}_{2t\text{-}TST(S)}$	$O(1)$	$O(zt)$	$O(n \log \sigma)$	$O(zt)$
Osoittimet	$O(1)$	$O(\frac{n}{t})$	$O(\frac{n}{t})$	$O(\frac{n}{t})$
Verkko G	-	-	$O(zt)$	$O(zt)$
Puu T, $\texttt{level\_ancestor}_T$	$O(1)$	$O(zt)$	$O(zt)$	$O(zt)$
t-lohkojen järjestäminen	-	-	$O(n \log \sigma)$	$O(zt)$
$code(S)$, $\texttt{lce}_{code(S)}$	$O(1)$	$O(\frac{n}{\sqrt{t}})$	$O(\frac{n}{\sqrt{t}})$	$O(\frac{n}{\sqrt{t}})$

varten saadaan vakioajassa solmun syvyys *depth*, jos syvyys on tallennettu kuhunkin puun T solmuun. Rivillä 10 minimin palauttava funktio on vakioaikainen, jos puun $2t$-$TST(S)$ jokaiseen solmuun on tallennettu tieto solmua vastaavan osajonon pituudesta.

Kysely $\texttt{long\_lce}_S$ palauttaa vastauksen ajassa $O(1)$. Kysely palautuu merkkijonolle $code(S)$ tehtäväksi vakioaikaiseksi kyselyksi $\texttt{lce}_{code(S)}$.

Täten kyselyn $\texttt{general\_lce}_S$ käytön aikavaativuus on $O(1)$. Kyselyjen $\texttt{short\_lce}_S$ ja $\texttt{long\_lce}_S$ lisäksi algoritmissa 2 (sivulla 36) käytetään tavallisia aritmeettisia operaatioita, sijoitusoperaatioita ja t-peitteen vakioaikaista funktiota $h(i, j)$.

Kyselyn $\texttt{short\_lce}_S$ käyttöön tarvitaan puu $2t$-$TST(S)$ apurakenteineen, virittävä puu T apurakenteineen ja noin $\frac{n}{t}$ osoitinta. Ne vievät tilaa yhteensä $O(zt + \frac{n}{t})$ konesanaa. Kyselyn $\texttt{long\_lce}_S$ käyttöön tarvitaan tilaa yhteensä $O(\frac{n}{\sqrt{t}})$ konesanaa, koska kyselyssä tarvitaan merkkijonoa $code(S)$ apurakenteineen.

Näin kyselyn $\texttt{general\_lce}_S$ käytön tilavaativuudeksi saadaan kyselyjen $\texttt{short\_lce}_S$ ja $\texttt{long\_lce}_S$ ja t-peitteen tilavaativuuksista $O(zt+\frac{n}{t})+O(\frac{n}{\sqrt{t}})+O(\frac{n}{\sqrt{t}}) = O(zt+\frac{n}{\sqrt{t}})$ konesanaa.

Kyselyn $\texttt{general\_lce}_S$ esikäsittelyn aikavaativuus on suurimmillaan $O(n \log \sigma)$ puulla $2t$-$TST(S)$ ja t-lohkoilla. Esikäsittelyn tilavaativuus on yhteensä $O(zt + \frac{n}{\sqrt{t}})$ konesanaa.

Tarkastellaan käytön tilavaativuuden $O(zt+\frac{n}{\sqrt{t}})$ minimointia. Lausekkeen derivaatta arvon t suhteen on

$$D_t\left(zt + \frac{n}{\sqrt{t}}\right) = z - \frac{n}{2}t^{-\frac{3}{2}}.$$

Laskemalla saadaan derivaatan nollakohdaksi $t = (\frac{n}{2z})^{\frac{2}{3}}$. Tilavaativuuden lausekkeen arvo välillä $1 \leq t \leq n$ on pienin nollakohdassa. Sijoittamalla se lausekkeeseen saadaan

$$zt + \frac{n}{\sqrt{t}} = z\left(\frac{n}{2z}\right)^{\frac{2}{3}} + n : \left(\frac{n}{2z}\right)^{\frac{2}{3}\frac{1}{2}} = 2^{-\frac{2}{3}} z^{\frac{1}{3}} n^{\frac{2}{3}} + 2^{\frac{1}{3}} z^{\frac{1}{3}} n^{\frac{2}{3}} = (2^{-\frac{2}{3}} + 2^{\frac{1}{3}}) z^{\frac{1}{3}} n^{\frac{2}{3}}.$$

Siis käytön tilavaativuudeksi tulee $O(z^{\frac{1}{3}} n^{\frac{2}{3}})$ konesanaa.

Derivaatan nollakohdan arvoa ei voi laskea suoraan, koska merkkijonoa S ei missään vaiheessa jaeta Lempel–Ziv 77 -tekijöihin eikä arvoa z siksi tunneta. Etsitään tilavaativuuden minimoiva arvo iteroimalla [50]. Asetetaan aluksi $t = 1$, ja kaksinkertaistetaan arvo joka kierroksella. Jokaisella arvolla t luodaan puu t-$TST(S)$, ja lasketaan yhteen sen todellinen koko ja arvo $n/\sqrt{t}$. Kun summa ensimmäisen kerran suurenee edellisestä, on löydetty minimin sisältävä väli. Jatketaan iterointia löydetyllä välillä puolittamalla väli joka kierroksella. Iteroinnissa on $O(\log n)$ kierrosta, mikä kasvattaa kyselyn $\texttt{general\_lce}_S$ esikäsittelyn aikavaativuutta kertoimella $\log n$ luokkaan $O(n \log \sigma \log n)$.

Koska käytön ja esikäsittelyn tilavaativuus on sama, myös esikäsittelyn tilavaativuudeksi tulee $O(z^{\frac{1}{3}} n^{\frac{2}{3}})$ konesanaa. Tämä ei ylity iteroidessakaan, koska puun t-$TST(S)$ tilavaativuus on luotaessa ja valmiina $O(zt)$ konesanaa ja optimaalista arvoa t lähestytään alhaalta.

Jos merkkijono S jaetaan toisistaan eroaviin tekijöihin jollakin tavalla, tekijöiden määrä on $O(n/\log_\sigma n)$, missä σ on aakkoston koko [40]. Käytettäessä merkkijonon Lempel–Ziv 77 -tekijöihinjakoa sama tekijä voi esiintyä useammin kuin kerran, mutta tekijöiden määrä kasvaa enintään vakiokertoimella.

Sijoittamalla $z = n/\log_\sigma n$ edelliseen tilavaativuuden lausekkeeseen saadaan [50]

$$z^{\frac{1}{3}} n^{\frac{2}{3}} = \left(\frac{n}{\log_\sigma n}\right)^{\frac{1}{3}} n^{\frac{2}{3}} = \frac{n}{(\log_\sigma n)^{\frac{1}{3}}} = n : \left(\frac{\log n}{\log \sigma}\right)^{\frac{1}{3}} = n\left(\frac{\log \sigma}{\log n}\right)^{\frac{1}{3}}.$$

Oletetaan $\sigma \leq 2^{o(\log n)}$, jolloin $\log \sigma \leq o(\log n)$. Nyt osamäärä $\frac{\log \sigma}{\log n}$ lähestyy nollaa, kun n kasvaa rajatta. Näin ollen tilavaativuudeksi tulee alilineaarinen $o(n)$ konesanaa, joka vastaa $o(n \log n)$ bittiä. Sijoittamalla oletus aiempaan esikäsittelyn aikavaativuustulokseen $O(n \log \sigma \log n)$ saadaan esikäsittelyn aikavaativuudeksi $o(n \log^2 n)$.

Taulukkoon 6 on koottu käsitellyt aika- ja tilavaativuudet. Kysely $\texttt{general\_lce}_S$ on kaikissa tapauksissa vakioaikainen. Kyselyn toteutus vie tilaa $O(zt + \frac{n}{\sqrt{t}})$ konesanaa, missä $1 \leq t \leq n$. Jos merkkijono S sisältää runsaasti toistoa, z on pieni suhteessa pituuteen n, ja termi zt on pieni suhteessa termiin $\frac{n}{\sqrt{t}}$. Tietorakenne voi tällöin viedä vähemmän tilaa kuin S. Jos z on paljon pienempi kuin n, arvo t voidaan valita riittävän suureksi suhteessa arvoon n, jotta tietorakenteen koko on alilineaarinen suhteessa merkkijonon S pituuteeen.

Jos t valitaan tilantarve minimoiden ja aakkosto on riittävän pieni, tietorakenteen

Taulukko 6: Kyselyn $\texttt{general\_lce}_S$ aika- ja tilavaativuuksia järjestetyn aakkoston $[1..\sigma]$ merkkijonolle $S[1..n]$. Tilavaativuudet on ilmaistu konesanoina. Oletetaan $zt < n$, missä z on merkkijonon S Lempel–Ziv 77 -tekijöiden määrä. Lisäksi ensimmäisen sarakkeen oletukset kumuloituvat, eli viimeisen rivin tulokset pätevät kaikkien oletusten täyttyessä.

Oletus	Käyttö		Esikäsittely	
	Aikav.	Tilav.	Aikav.	Tilav.
$1 \leq t \leq n$	$O(1)$	$O(zt + \frac{n}{\sqrt{t}})$	$O(n \log \sigma)$	$O(zt + \frac{n}{\sqrt{t}})$
$t = (\frac{n}{z})^{\frac{2}{3}}$	$O(1)$	$O(z^{\frac{1}{3}} n^{\frac{2}{3}})$	$O(n \log \sigma \log n)$	$O(z^{\frac{1}{3}} n^{\frac{2}{3}})$
$\sigma \leq 2^{o(\log n)}$	$O(1)$	$o(n)$	$o(n \log^2 n)$	$o(n)$

koko on alilineaarinen suhteessa merkkijonon S pituuteen. Tämä koskee myös merkkijonoja, jotka eivät tiivisty Lempel–Ziv 77 -tekijöihinjaon avulla.

Tarkastellaan vielä aika- ja tilavaativuutta pienellä aakkostolla [50], jolla konesanaan mahtuu useita alkuperäisen merkkijonon S merkkejä. Jos jokainen aakkoston merkki vie tilaa $\log \sigma$ bittiä, konesanaan pituudeltaan $\log n$ bittiä mahtuu $\frac{\log n}{\log \sigma} = \log_\sigma n$ merkkiä. Käsitellään merkkijonoa S bittijonona, jonka pituus on $n \log \sigma$ bittiä.

Olkoon $t = \log n$. Nyt vakioaikainen kysely $\texttt{short\_lce}_S$ ei tarvitse lainkaan tietorakenteita. Kahden bittijonon, joiden pituudet ovat $\log n$ bittiä, pisin yhteinen alkuosa saadaan bittikohtaisella *xor*-operaatiolla. Jos tuloksessa on pelkkiä `0`-bittejä, jonot ovat samat. Muuten tuloksen merkitsevin `1`-bitti on samassa kohdassa kuin jonojen ensimmäinen toisistaan eroava bitti. Kun tulos tulkitaan kokonaisluvuksi k ja vähiten merkitsevän bitin indeksi on nolla, eroavan bitin indeksi on $\lfloor \log k \rfloor$, josta helposti lasketaan pisimmän yhteisen alkuosan pituus.

Vakioaikainen kysely $\texttt{long\_lce}_S$ suoritetaan bittijonolle samalla tavalla kuin normaalisti merkkijonolle S. Tarvittavat tietorakenteet vievät normaalisti tilaa $O(\frac{n}{\sqrt{t}})$ konesanaa eli $O(\frac{n}{\sqrt{t}} \log n)$ bittiä. Sijoittamalla pituus $n \log \sigma$ ja $t = \log n$ lausekkeeseen saadaan

$$\frac{n \log \sigma}{\sqrt{\log n}} \log(n \log \sigma) = n \sqrt{\log n} \log \sigma \frac{\log(n \log \sigma)}{\log n} = n \sqrt{\log n} \log \sigma \left(1 + \frac{\log \log \sigma}{\log n}\right).$$

Viimeinen suluissa oleva summa lähestyy arvoa 1, kun n kasvaa rajatta. Siis tilavaativuudeksi tulee $O(n \sqrt{\log n} \log \sigma)$ bittiä.

Oletetaan $\sigma \leq 2^{o(\sqrt{\log n})}$, jolloin $\log \sigma \leq o(\sqrt{\log n})$. Nyt tilavaativuudeksi tulee $o(n \sqrt{\log n} \sqrt{\log n}) = o(n \log n)$ bittiä. Koska kysely on vakioaikainen, aika- ja tilavaativuuden tulo on $o(n \log n)$. Tämä on alempi kuin luvussa 4.3 käsitelty tiukka alaraja $\Omega(n \log n)$. Ristiriitaa ei silti ole. Alarajaa osoitettaessa oletettiin muistialkion sisältävän vain yhden syötteen merkin, kun taas nyt konesanaan oletetaan mahtuvan useita merkkejä.

Kun algoritmi suoritetaan bittijonoksi tulkitulle merkkijonolle, tarvitaan alkuperäinen merkkijono S, kun taas normaalisti algoritmi käyttää siitä luotua merkkijonoa

S'. Tilavaativuuden näkökulmasta bittijonoksi tulkitseminen voi kannattaa, jos σ on riittävän pieni ja S sisältää vain vähän toistoa. Tarpeeksi toisteiselle merkkijonolle S puolestaan pätee $zt < n$, jolloin tilavaativuus voi muodostua pienemmäksi normaalissa suorituksessa.

5.6 Tietorakenteen arviointia

Edellisissä alaluvuissa kuvattiin kyselyn $\texttt{general\_lce}_S$ tietorakenne ja sen aika- ja tilavaativuudet. Tietorakenteesta voi palauttaa alkuperäisen merkkijonon, mikä näytetään seuraavaksi. Sen jälkeen minimoidaan puun $2t$-$TST(S)$ tilavaativuutta puuhun tehtävillä muutoksilla. Niiden jälkeen palauttaminen ei enää onnistu informaation vähenemisen vuoksi. Lisäksi esitetään puun tilavaativuus merkkijonoattraktorin koon avulla.

Olkoon $S[1..n]$ järjestetyn aakkoston merkkijono, josta on luotu tietorakenne kyselyä $\texttt{general\_lce}_S$ varten. Nyt puussa $2t$-$TST(S)$ jokaista joukon $Subs_{2t}(S)$ merkkijonoa vastaa polku juuresta lehteen. Lisäksi on tallennettu osoittimet, jotka viittaavat lehtiin $l(1)$, $l(1+t)$, $l(1+2t)$, $l(1+3t)$ ja niin edelleen. Polku juuresta lehteen $l(i)$ vastaa osajonoa $S[i..\min(i+2t-1, n)]$. Merkkijono S saadaan selville tietorakenteesta merkitsemällä ensin arvo i jokaiseen lehteen $l(i)$, johon viittaa osoitin. Sitten puun $2t$-$TST(S)$ polut juuresta lehteen käydään läpi. Jos lehteen on merkitty yksi tai useampi arvo i, löytyi osajono, jonka paikka tai paikat merkkijonossa S tiedetään. Osajonon voi lukea kyseisen polun kaarten viitteiden avulla. Osajonoista voi koota alkuperäisen merkkijonon S.

Tietorakenne sisältää siis merkkijonon S kaiken informaation, mutta vähempikin informaatio riittää LCE-ongelman ratkaisemiseen. Kyselyyn $\texttt{short\_lce}_S$ vastaamiseen ei tarvita puun $2t$-$TST(S)$ kaarten viitteitä ja merkkijonoa S'. Riittää tallentaa tieto juuresta solmuun johtavaa polkua vastaavan osajonon pituudesta jokaiseen solmuun, jolle pituus on alle t. Tietoa tätä pitemmistä poluista ei tarvita solmuihin, koska kysely palauttaa enintään arvon t. Jos kahden lehden alin yhteinen esisä on solmu, johon pituutta ei ole merkitty, voidaan palauttaa arvo t. Puussa voi muutoksen jälkeen olla polkuja, joissa on vähintään kolme peräkkäistä merkitsemätöntä solmua. Poluista voi poistaa merkitsemättömät solmut, jotka ovat lehden ja siihen johtavan polun ylimmän merkitsemättömän solmun välissä. Lehti siirretään ylimmän merkitsemättömän solmun lapseksi. Lopputuloksena saatu puu riittää yhä kyselyyn $\texttt{short\_lce}_S$ vastaamiseen.

Muutosten jälkeen puun $2t$-$TST(S)$ tilavaativuusluokka on sama $O(zt)$ konesanaa kuin alkuperäisen puun, koska puussa on edelleen $O(zt)$ lehteä ja siten $O(zt)$ solmua. Puu voi kuitenkin viedä vähemmän tilaa. Puun muokkaaminen ei muuta esikäsittelyn aika- tai tilavaativuutta. Koska merkkijono S' ei ole välttämätön puuta $2t$-$TST(S)$ käytettäessä, puu on mahdollista luoda myös suoraan merkkijonosta S. Vakioaakkostolla ja kokonaislukuaakkostolla tämä vie aikaa $O(n)$ [39].

Muiden kyselyn $\texttt{general\_lce}_S$ rakenteiden viemän tilan vähentämiseen ei ole ilmeisiä keinoja. Merkkijono $code(S)$ sisältää tiedot t-lohkojen aakkosjärjestyksestä ja

sijainnista, mikä on tarpeen kyselyyn $\texttt{long\_lce}_S$. Vakioaikaiseen funktioon $h(i,j)$ tarvitaan t-peitettä. Puun T toteutuksessa solmuihin ei tarvitse tallentaa osajonoja tai kaariin nimeäviä merkkejä, sillä kyselyyn $\texttt{level\_ancestor}_T$ riittää puun rakenne.

Jos puu $2t$-$TST(S)$ ei sisällä kaarten viitteitä, kyselyn $\texttt{general\_lce}_S$ tietorakenne on koodaava. Alkuperäiseen merkkijonoon S päästään käsiksi vain kyselyn avulla. Merkkijonoa S ei tarvita tietorakenteen luomisen jälkeen, eikä sitä voi palauttaa. Tietorakenne eroaa näin aiemmista LCE-ongelman tilaa säästävistä ratkaisuista, joista osa esiteltiin luvussa 4.3.

Puun $2t$-$TST(S)$ ja virittävän puun T voi perinteisen puun sijasta toteuttaa myös tasapainoisia sulkeita kuvaavana bittivektorina. Bittivektorit voi luoda valmiista puista käymällä ne läpi syvyysjärjestyksessä. Puun $2t$-$TST(S)$ bittivektori sisältää riittävästi informaatiota alimman yhteisen esi-isän etsintään eli kyselyyn `lca`, koska puun $2t$-$TST(S)$ kaarten viitteitä ei tarvita vastaamiseen. Puun T bittivektori puolestaan sisältää riittävästi informaatiota kyselyyn `level_ancestor`. Jos kumpaakin bittivektoria täydennetään rmM-puulla, bittivektoreille voi toteuttaa kyselyt `lca` ja `level_ancestor` [11].

Perinteiset puut vievät tilaa $O(zt)$ konesanaa, mutta tätä alempaan luokkaan bittivektoreilla ei kuitenkaan päästä. Bittivektorit vievät tilaa $O(zt)$ bittiä ja kyselyjen apurakenteet $o(zt)$ bittiä, mutta lisäksi kyselyssä $\texttt{short\_lce}_S$ tarvitaan vakioaikainen siirtyminen mistä tahansa puun T solmusta sitä vastaavaan puun $2t$-$TST(S)$ lehteen. Bittivektorissa solmu yksilöidään sulkevalla sululla eli `0`-bitillä, joka löydetään esimerkiksi `select`-operaatiolla. Ei kuitenkaan ole tiedossa yleistä laskukaavaa, jolla saadaan yksittäistä `0`-bittiä, joka on puuta T kuvaavassa bittivektorissa, vastaavan `0`-bitin paikka puuta $2t$-$TST(S)$ kuvaavassa bittivektorissa. Jokaisen solmun vastinsolmun paikka on tallennettava erikseen, mikä vie tilaa yhteensä $O(zt \log zt)$ bittiä eli $O(zt)$ konesanaa. Näin tilavaativuus nousee samaan luokkaan kuin perinteisillä puilla.

Tietorakenteen kehittäjät ehdottavat tutkittavaksi puun $2t$-$TST(S)$ tilavaativuuden ylärajan parantamista tiukemmaksi kuin $O(zt)$ konesanaa, sillä zt on jossakin tapauksessa luokkaa $\sqrt{n}$ suurempi kuin puun tosiasiallinen koko [50]. Esitetään seuraavaksi yläraja vastikään kehitetyn merkkijonoattraktorin [33] avulla. Attraktori on tiivistämisen sanakirjamenetelmien teoriaa yhtenäistävä käsite, jolla voi mitata merkkijonon erilaisten osajonojen määrää ja siten merkkijonon toisteisuutta.

Joukko kokonaislukuja $\Gamma = \{j_1, \ldots, j_\gamma\}$ on merkkijonon $S[1..n]$ merkkijonoattraktori, jos jokaisella osajonolla $S[i..j]$ on esiintymä $S[i'..j'] = S[i..j]$ siten, että $j_k \in [i', j']$ jollakin $j_k \in \Gamma$. Esimerkiksi $\Gamma = \{4, 7, 11, 12\}$ on merkkijonon `cdabccdabcca` attraktori. Joukon alkioita vastaavat merkkijonon merkit on alleviivattu. Attraktorin koko $\gamma = 4$, ja attraktori on pienin mahdollinen esimerkin merkkijonolle. Jos joukkoon Γ lisätään kokonaislukuja, se on edelleen kyseisen merkkijonon attraktori.

Osoitetaan puun $2t$-$TST(S)$ tilavaativuudeksi $O(\gamma^* t)$ konesanaa, missä γ^* on merkkijonon $S[1..n]$ pienimmän merkkijonoattraktorin koko. Olkoon Γ^* tällainen pienin merkkijonoattraktori. Määritelmän nojalla jokaisen osajonon $S[i..j]$ jollekin esiinty-

mälle $S[i'..j']$ on olemassa merkki $S[j_k]$ siten, että $j_k \in \Gamma^*$ ja $i' \leq j_k \leq j'$. Erityisesti tämä pätee kaikille merkkijonon S osajonoille $S[i..i+q-1]$ pituudeltaan q. Toisaalta jokaista tällaista merkkiä $S[j_k]$ kohti on olemassa enintään q erilaista osajonoa pituudeltaan q, jotka kattavat merkin $S[j_k]$. Siis merkkijonossa S on enintään $\gamma^* q$ erilaista merkkijonoa pituudeltaan q. Näin ollen $|Subs_q(S)| = O(\gamma^* q)$. Koska puussa q-$TST(S)$ on $|Subs_q(S)|$ lehteä, puun tilavaativuus on $O(\gamma^* q)$ konesanaa. Siis puun $2t$-$TST(S)$ tilavaativuus on $O(\gamma^* t)$ konesanaa. Sama pätee virittävän puun T tilavaativuuteen, koska puun T solmut vastaavat yksi yhteen puun $2t$-$TST(S)$ lehtiä.

Merkkijonon $S[1..n]$ Lempel–Ziv 77 -tekijöihinjaon voi tulkita approksimointialgoritmiksi, jonka tuloksena saatava tekijöiden määrä z on arvio merkkijonon pienimmän attraktorin koolle [33]. Merkkijonolla S on nimittäin aina attraktori kooltaan z. Siis $z \geq \gamma^*$. Tästä ja äsken osoitetusta tilavaativuudesta $O(\gamma^* q)$ konesanaa seuraa suoraan jo luvussa 5.4 osoitettu tulos, että puun q-$TST(S)$ tilavaativuus on $O(zq)$ konesanaa. Lisäksi tekijöiden määrälle on osoitettu yläraja $z \in O(\gamma^* \log^2(n/\gamma^*))$. Ei kuitenkaan ole tiedossa, onko tämä yläraja tiukka. Näin ollen on avoin kysymys, onko $O(\gamma^* t)$ konesanaa puun $2t$-$TST(S)$ tilavaativuutena asymptoottisesti tiukempi kuin $O(zt)$ konesanaa.

Pienimmän attraktorin määrittäminen laskennallisesti on NP-täydellinen ongelma, jos attraktorissa tarkasteltavien osajonojen pituus $k \geq 3$. Tämä on osoitettu tarkastelemalla k-attraktoria, jossa osajonojen pituus on enintään k. Edellä määritelty merkkijonoattraktori on tällöin erikoistapaus, jossa osajonojen pituus on enintään n. Jos $k \geq 3$, myös pienimmän k-tarkan attraktorin, jossa tarkastellaan vain osajonoja pituudeltaan täsmälleen k, määrittäminen on NP-täydellinen ongelma [32]. Siten merkkijonon S pienintä attraktoria ei ilmeisesti voi laskea polynomisessa ajassa. Sen kokoa voi kuitenkin arvioida Lempel–Ziv 77 -tekijöihinjaon avulla, joka onnistuu lineaarisessa ajassa merkkijonon S pituuteen nähden.

Kyselyn $\texttt{general\_lce}_S$ aikavaativuus on $O(1)$, mutta kyselyn nopeudesta sovelluksissa ei ole saatavilla tietoa. Aikavaativuuden vakiokerroin riippuu pitkälti kyselyistä $\texttt{short\_lce}_S$ ja $\texttt{long\_lce}_S$. Näiden toteutukset nojaavat vakioaikaisuuden saavuttamiseksi apurakenteisiin $\texttt{level\_ancestor}_T$, $\texttt{lca}_{2t\text{-}TST(S)}$ ja $\texttt{lce}_{code(S)}$. Apurakenteiden toteutus vaikuttaa todennäköisesti eniten kyselyn $\texttt{general\_lce}_S$ käytännön nopeuteen.

Laajempi avoin kysymys on, miten tilaa säästävien LCE-ongelman ratkaisujen nopeus ja tilantarve vaihtelevat ratkaisujen välillä erilaisilla merkkijonoilla. Viime vuosina on kehitetty useita ratkaisuja, mutta niiden toimivuudesta sovelluksissa ei ole vertailutietoa. Nopeusvertailuun on perusteltua ottaa mukaan tietorakenteeton ratkaisu, sillä luvussa 2.2 kerrotusti se on käytännössä nopea, jos loppuosien pisimmät yhteiset alkuosat ovat keskimäärin lyhyitä.

6 Johtopäätökset

Merkkijonon $S[1..n]$ kahden loppuosan pisimmän yhteisen alkuosan pituuden selvittäminen eli kyselyyn $\texttt{lce}_S(i, j)$ vastaaminen on keskeinen osaongelma monissa merkkijonoalgoritmeissa. Jos aakkoston merkit ovat riippumattomia ja samoin jakautuneita, loppuosien yhteiset alkuosat ovat keskimäärin lyhyitä. Mutta esimerkiksi luonnollisissa kielissä ja DNA-juosteessa maksimipituudet ovat selvästi suurempia, jolloin merkkijonosta S kannattaa useita kyselyjä tehtäessä luoda tietorakenne vastaamista varten.

LCE-ongelman voi ratkaista ilman tilaa säästäviä tietorakenteita loppuosataulukon tai -puun avulla. Rakenteet voi luoda ja täydentää apurakenteilla vakioaikaista kyselyä varten ajassa $O(n)$, jos aakkosto on vakio- tai kokonaislukuaakkosto. Tietorakenteiden tilavaativuus on $O(n)$ konesanaa, mutta sovelluksissa ne vievät usein tilaa moninkertaisesti merkkijonoon S verrattuna.

Tiedon määrän kasvu on korostanut tarvetta minimoida tietorakenteiden kokoa, jotta ne mahtuvat keskusmuistiin ja nopeisiin välimuisteihin. Tiedon voi esittää informaation vähenemättä alkuperäistä tiiviimmin, jos se sisältää toistoa tai aakkosto on koodattu tarpeettoman monella bitillä. Tilantarpeen alarajan voi laskea informaatioteoreettisen entropian avulla. Laskemiseen on useita malleja, esimerkiksi pahimman tapauksen entropia ja Shannon-entropia. Merkkijonoille yksi tiivistyvyyden mitta on myös Lempel–Ziv-tekijöiden määrä. Häviöttömässä tiivistyksessä ja entropiakoodauksessa pyritään lähestymään jollakin mallilla laskettua tiedon entropiaa.

Tilaa säästävien tietorakenteiden luomiseen on kehitetty perinteisestä tiivistyksestä eroavia keinoja, jotta rakenteet tukevat haluttuja operaatioita ilman purkamista. Tietorakenteiden toteutuksessa on olennaista esittää taulukot ja puut tiiviisti. Etenkin järjestyspuiden, joissa on n solmua, pahimman tapauksen entropia $2n - \Theta(\log n)$ sallii puun toteutuksen perinteistä $O(n \log n)$ bitin kokoa tiiviimmin. Puun esittämiseen noin $2n$ bittiä käyttäen onkin olemassa kolme keskeistä tapaa. Tasapainoisia sulkeita kuvaava bittivektori vaikuttaa niistä monikäyttöisimmältä sovelluksiin, sillä $o(n)$ bitin apurakenteella se tukee useita puun operaatioita.

Operaatioissa ei aina tarvita kaikkea alkuperäistä tietoa. Tyypillisiä esimerkkejä ovat osavälikyselyn muunnelmat, joissa vastaukseksi riittää taulukon alkion indeksi alkion sijasta. Tällöin kyselyihin samat vastaukset palauttavat taulukon permutaatiot ovat keskenään ekvivalentteja, joten kyselyjen toteutukseen riittävät tiedot alkioiden keskinäisestä suuruusjärjestyksestä. Toteutuksessa informaatio vähenee, ja tietorakenne voi olla kooltaan alilineaarinen suhteessa alkuperäiseen taulukkoon. Tällaisesta koodaavasta tietorakenteesta ei voi palauttaa alkuperäistä tietoa, vaan tietoon päästään käsiksi vain kyselyjen kautta.

Esimerkki koodaavasta tietorakenteesta on osavälin pienimmän alkion indeksin palauttavan kyselyn toteutus taulukolle, jossa on n kokonaislukua. Tietorakenteen vakioaikainen toteutus $2n + o(n)$ bittiä käyttäen perustuu bittivektoriin, joka kuvaa taulukkoa vastaavasta karteesisesta puusta luotua järjestyspuuta. Bittivektoria täydennetään tarvittavat puun operaatiot toteuttavalla apurakenteella. Kyselyn toteu-

tus havainnollistaa tilaa säästävissä rakenteissa yleisiä tekniikoita, joilla tavoitellaan optimaalista aika–tila-vaihtokauppaa. Teoriassa nopein toteutus ei aina ole käytännössä nopein samaan tilavaativuusluokkaan kuuluvista toteutuksista.

Jos merkkijonon $S[1..n]$ kysely $\texttt{lce}_S$ toteutetaan hajasaantimallissa $O(n/t)$ bitin systemaattisella tietorakenteella, missä $1 \leq t \leq n$, kyselyn aikavaativuudeksi on osoitettu tietyin edellytyksin $\Omega(t)$. Tällöin aika- ja tilavaativuuden tulo on $\Omega(n)$. Mallissa, jossa aikavaativuuden mittaus perustuu muistialkioiden luku- ja kirjoituskertoihin, on puolestaan osoitettu aika- ja tilavaativuuden tuloksi tiukka $\Omega(n \log n)$. Sopivalla algoritmilla voidaan silti saavuttaa alempi vaativuuksien tulo, jos algoritmi ei täytä kaikkia tulokseen liittyviä oletuksia.

LCE-ongelmaan on olemassa useita tilaa säästäviä ratkaisuja, joiden aika- ja tilavaativuudet vaihtelevat. Ratkaisuissa käytetään hyväksi esimerkiksi t-peitettä, Karp–Rabin-tunnisteita tai sopivaa kielioppia. Osa ratkaisuista on deterministisiä, kun taas osa palauttaa oikean vastauksen suurella todennäköisyydellä.

LCE-ongelman voi ratkaista myös koodaavalla tietorakenteella. Ratkaisun toteutus on jaettu kahteen osaan. Ensimmäinen palauttaa arvoa t lyhyemmät kyselyn $\texttt{lce}_S(i, j)$ vastaukset. Toteutus perustuu merkkijonon S tietyistä osajonoista luotuun loppuosapuuhun, josta etsitään kahta loppuosaa vastaavien lehtien alin yhteinen esi-isä. Lisäksi käytetään de Bruijn -verkon muunnosta, virittävää puuta ja osoittimia. Toinen osa palauttaa tiedon, montako arvon t pituista yhteistä osajonoa kahden loppuosan alussa on. Tätä varten merkkijonosta S luodaan lyhyempi merkkijono, jossa merkkijonon S osajonoja pituudeltaan t käsitellään yhtenä merkkinä. Näin $\texttt{lce}_S(i, j)$-kysely palautuu luodun merkkijonon vastaavaksi kyselyksi.

Tämä LCE-ongelman ratkaisu on vakioaikainen ja vie tilaa $O(zt + \frac{n}{\sqrt{t}})$ konesanaa, missä z on merkkijonon S Lempel–Ziv 77 -tekijöiden määrä. Esikäsittelyn aikavaativuus on $O(n \log \sigma)$, missä σ on aakkoston koko, ja tilavaativuus $O(zt + \frac{n}{\sqrt{t}})$ konesanaa. Tilavaativuuden näkökulmasta ratkaisu on kiinnostava verrattuna perinteisiin LCE-ongelman ratkaisuihin, jos $zt < n$. Jos arvo t valitaan tilavaativuus minimoiden, tilavaativuudeksi tulee riittävän pienellä aakkostolla $o(n)$ konesanaa eli $o(n \log n)$ bittiä. Tällöin esikäsittelyn aikavaativuus kuitenkin kasvaa. Tilavaativuus voi aakkostosta riippumatta olla alilineaarinen myös, jos merkkijono S on hyvin tiivistyvä Lempel–Ziv-tekijöihinjaon avulla.

Kyseessä on ensimmäinen LCE-ongelman vakioaikainen ratkaisu, jossa alkuperäistä merkkijonoa ei tarvita tietorakenteen luomisen jälkeen ja jolla voidaan päästä alilineaariseen tilavaativuuteen ilman merkkijonon S korkeaa tiivistyvyyttä. Perinteisen puurakenteen voi ratkaisussa korvata tasapainoisia sulkeita kuvaavalla bittivektorilla, mutta muutos ei kokonaisuus huomioon ottaen johda pienempään tilavaativuusluokkaan. Lisäksi tilavaativuuden termin zt voi kirjoittaa toisin merkkijonoattraktorin koon avulla, mutta tuloksena olevan tilavaativuuden ei ole osoitettu olevan asymptoottisesti pienempi.

Viime vuosina kehitettyjen tilaa säästävien LCE-ongelman ratkaisujen toimivuudesta sovelluksissa ei ole olemassa vertailua.

Lähteet

1 Abeliuk, Andrés, Cánovas, Rodrigo ja Navarro, Gonzalo: *Practical compressed suffix trees.* Algorithms, 6(2):319–351, 2013.

2 Apostolico, Alberto, Crochemore, Maxime, Farach-Colton, Martin, Galil, Zvi ja Muthukrishnan, S.: *40 years of suffix trees.* Commun. ACM, 59(4):66–73, 2016.

3 Bender, Michael A. ja Farach-Colton, Martín: *The LCA problem revisited.* Teoksessa *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium 2000 Proceedings*, sivut 88–94, 2000.

4 Bender, Michael A. ja Farach-Colton, Martín: *The level ancestor problem simplified.* Theor. Comput. Sci., 321(1):5–12, 2004.

5 Benoit, David, Demaine, Erik D., Munro, J. Ian, Raman, Rajeev, Raman, Venkatesh ja Rao, S. Srinivasa: *Representing trees of higher degree.* Algorithmica, 43(4):275–292, 2005.

6 Bille, Philip, Cording, Patrick Hagge, Fischer, Johannes ja Gørtz, Inge Li: *Lempel-Ziv compression in a sliding window.* Teoksessa *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, sivut 15:1–15:11, 2017.

7 Bille, Philip, Gawrychowski, Pawel, Gørtz, Inge Li, Landau, Gad M. ja Weimann, Oren: *Longest common extensions in trees.* Theor. Comput. Sci., 638:98–107, 2016.

8 Bille, Philip, Gørtz, Inge Li, Sach, Benjamin ja Vildhøj, Hjalte Wedel: *Time–space trade-offs for longest common extensions.* J. Discrete Algorithms, 25:42–50, 2014.

9 Burkhardt, Stefan ja Kärkkäinen, Juha: *Fast lightweight suffix array construction and checking.* Teoksessa *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003 Proceedings*, sivut 55–69, 2003.

10 Clark, David: *Compact pat trees.* Väitöskirja, University of Waterloo, 1996.

11 Cordova, Joshimar ja Navarro, Gonzalo: *Simple and efficient fully-functional succinct trees.* Theor. Comput. Sci., 656:135–145, 2016.

12 Davoodi, Pooya, Raman, Rajeev ja Satti, Srinivasa Rao: *On succinct representations of binary trees.* Mathematics in Computer Science, 11(2):177–189, 2017.

13 Duda, Jarek, Tahboub, Khalid, Gadgil, Neeraj J. ja Delp, Edward J.: *The use of asymmetric numeral systems as an accurate replacement for Huffman coding.* Teoksessa *2015 Picture Coding Symposium, PCS 2015*, sivut 65–69, 2015.

14 Farach-Colton, Martin, Ferragina, Paolo ja Muthukrishnan, S.: *On the sorting-complexity of suffix tree construction.* J. ACM, 47(6):987–1011, 2000.

15 Ferrada, Héctor ja Navarro, Gonzalo: *Improved range minimum queries.* J. Discrete Algorithms, 43:72–80, 2017.

16 Fischer, Johannes: *Inducing the LCP-array.* Teoksessa *Algorithms and Data Structures - 12th International Symposium, WADS 2011 Proceedings*, sivut 374–385, 2011.

17 Fischer, Johannes ja Heun, Volker: *Space-efficient preprocessing schemes for range minimum queries on static arrays.* SIAM J. Comput., 40(2):465–492, 2011.

18 Gabow, Harold N., Bentley, Jon Louis ja Tarjan, Robert E.: *Scaling and related techniques for geometry problems.* Teoksessa *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, 1984*, sivut 135–143, 1984.

19 Gawrychowski, Pawel, Kociumaka, Tomasz, Rytter, Wojciech ja Waleń, Tomasz: *Faster longest common extension queries in strings over general alphabets.* Teoksessa *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, sivut 5:1–5:13, 2016.

20 Gawrychowski, Pawel ja Nicholson, Patrick K.: *Optimal query time for encoding range majority.* Teoksessa *Algorithms and Data Structures - 15th International Symposium, WADS 2017 Proceedings*, sivut 409–420, 2017.

21 Gog, Simon: *Compressed suffix trees: design, construction, and applications.* Väitöskirja, University of Ulm, 2011.

22 Golin, Mordecai, Iacono, John, Krizanc, Danny, Raman, Rajeev, Satti, Srinivasa Rao ja Shende, Sunil: *Encoding 2D range maximum queries.* Theoretical Computer Science, 609:316–327, 2016.

23 Gusfield, Dan: *Algorithms on strings, trees, and sequences - computer science and computational biology.* Cambridge University Press, 1997.

24 Hoffmann, Michael, Iacono, John, Nicholson, Patrick K. ja Raman, Rajeev: *Encoding nearest larger values.* Theor. Comput. Sci., 710:97–115, 2018.

25 Howard, Paul G. ja Vitter, Jeffrey Scott: *Arithmetic coding for data compression.* Proceedings of the IEEE, 82(6):857–865, 1994.

26 Huffman, David A.: *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 40(9):1098–1101, 1952.

27 I, Tomohiro: *Longest common extensions with recompression.* Teoksessa *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, sivut 18:1–18:15, 2017.

28 Ilie, Lucian, Navarro, Gonzalo ja Tinta, Liviu: *The longest common extension problem revisited and applications to approximate string searching.* J. Discrete Algorithms, 8(4):418–428, 2010.

29 Jacobson, Guy: *Space-efficient static trees and graphs.* Teoksessa *30th Annual Symposium on Foundations of Computer Science, 1989*, sivut 549–554, 1989.

30 Kärkkäinen, Juha ja Kempa, Dominik: *Engineering external memory LCP array construction: parallel, in-place and large alphabet.* Teoksessa *16th International Symposium on Experimental Algorithms, SEA 2017*, sivut 17:1–17:14, 2017.

31 Kärkkäinen, Juha, Kempa, Dominik ja Puglisi, Simon J.: *Hybrid compression of bitvectors for the FM-index.* Teoksessa *Data Compression Conference, DCC 2014*, sivut 302–311, 2014.

32 Kempa, Dominik, Policriti, Alberto, Prezza, Nicola ja Rotenberg, Eva: *String attractors: verification and optimization.* Teoksessa *26th Annual European Symposium on Algorithms, ESA 2018*, sivut 52:1–52:13, 2018.

33 Kempa, Dominik ja Prezza, Nicola: *At the roots of dictionary compression: string attractors.* Teoksessa *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018*, sivut 827–840, 2018.

34 Knuth, Donald E.: *The art of computer programming, volume 4A: combinatorial algorithms, part 1.* Addison Wesley, 1. painos, 2011.

35 Knuth, Donald E., Morris, Jr., James H. ja Pratt, Vaughan R.: *Fast pattern matching in strings.* SIAM J. Comput., 6(2):323–350, 1977.

36 Kosolobov, Dmitry: *Tight lower bounds for the longest common extension problem.* Inf. Process. Lett., 125:26–29, 2017.

37 Manber, Udi ja Myers, Gene: *Suffix arrays: a new method for on-line string searches.* SIAM J. Comput., 22(5):935–948, 1993.

38 Munro, J. Ian ja Raman, Venkatesh: *Succinct representation of balanced parentheses and static trees.* SIAM J. Comput., 31(3):762–776, 2001.

39 Na, Joong Chae, Apostolico, Alberto, Iliopoulos, Costas S. ja Park, Kunsoo: *Truncated suffix trees and their application to data compression.* Theor. Comput. Sci., 1-3(304):87–101, 2003.

40 Navarro, Gonzalo: *Compact data structures: a practical approach.* Cambridge University Press, 2016.

41 Navarro, Gonzalo ja Sadakane, Kunihiko: *Fully functional static and dynamic succinct trees.* ACM Trans. Algorithms, 10(3):16:1–16:39, 2014.

42 Okanohara, Daisuke ja Sadakane, Kunihiko: *Practical entropy-compressed rank/select dictionary.* Teoksessa *Proceedings of the Meeting on Algorithm Engineering & Experiments, ALENEX 2007*, sivut 60–70, 2007.

43 Pagh, Rasmus: *Low redundancy in static dictionaries with constant query time.* SIAM J. Comput., 31(2):353–363, 2001.

44 Prezza, Nicola: *In-place sparse suffix sorting.* Teoksessa *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, sivut 1496–1508, 2018.

45 Raman, Rajeev: *Encoding data structures.* Teoksessa *WALCOM: Algorithms and Computation - 9th International Workshop, 2015*, sivut 1–7, 2015.

46 Raman, Rajeev ja Rao, S. Srinivasa: *Succinct representations of ordinal trees.* Teoksessa *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, sivut 319–332, 2013.

47 Schwartz, Eugene S. ja Kallick, Bruce: *Generating a canonical prefix encoding.* Commun. ACM, 7(3):166–169, 1964.

48 Shannon, C. E.: *A mathematical theory of communication.* The Bell System Technical Journal, 27:379–423, 623–656, 1948.

49 Skala, Matthew: *Array range queries.* Teoksessa *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, sivut 333–350, 2013.

50 Tanimura, Yuka, Nishimoto, Takaaki, Bannai, Hideo, Inenaga, Shunsuke ja Takeda, Masayuki: *Small-space LCE data structure with constant-time queries.* Teoksessa *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017*, sivut 10:1–10:15, 2017.

51 Ukkonen, E.: *On-line construction of suffix trees.* Algorithmica, 14(3):249–260, 1995.

52 Vitale, Luciana, Martín, Álvaro ja Seroussi, Gadiel: *Space-efficient representation of truncated suffix trees, with applications to Markov order estimation.* Theor. Comput. Sci., 595:34–45, 2015.

53 Vuillemin, Jean: *A unifying look at data structures.* Commun. ACM, 23(4):229–239, 1980.

54 Weiner, Peter: *Linear pattern matching algorithms.* Teoksessa *14th Annual Symposium on Switching and Automata Theory, 1973*, sivut 1–11, 1973.

55 Zhou, Dong, Andersen, David G. ja Kaminsky, Michael: *Space-efficient, high-performance rank and select structures on uncompressed bit sequences.* Teoksessa *Experimental Algorithms, 12th International Symposium, SEA 2013 Proceedings*, sivut 151–163, 2013.

56 Ziv, Jacob ja Lempel, Abraham: *A universal algorithm for sequential data compression.* IEEE Trans. Information Theory, 23(3):337–343, 1977.